Agint: Agentic Graph Compilation for Software Engineering Agents

Abhi Chivukula
Agint
Abhi@AgintAI.com

Jay Somasundaram
Agint
Jay@AgintAI.com

Vijay Somasundaram
Agint
Vijay@AgintAI.com

Abstract

LLM-based coding agents are increasingly common, but still face challenges in context management, latency, reliability, reproducibility, and scalability. We present **Agint**, an agentic graph compiler, interpreter, and runtime that incrementally and hierarchically converts natural-language instructions into typed, *effect-aware* code DAGs (Directed Acyclic Graphs). Agint introduces explicit *type floors* (TEXT $\rightarrow$ TYPED $\rightarrow$ SPEC $\rightarrow$ CODE) grounded in semantic graph transformations and a hybrid LLM/function JIT runtime, enabling dynamic graph refinement, reproducible and optimizable execution, speculative evaluation, and interoperability with existing developer tools. Agint's typed graph bindings ensure reliability and enable *concurrent composition of codebases by construction*, supporting accelerated development using smaller, faster models for lower latency, more efficient context use, and higher throughput. Hierarchical compilation allows scalable graph edits, while the graph structure provides reproducibility and efficient parallel generation. Agint provides a composable Unix-style toolchain: **dagify** *(DAG compiler)*, **dagent** *(hybrid JIT runtime)*, **schemagin** *(schema generator)*, and **datagin** *(data transformer)* for realtime, low-latency code and dataflow creation. Human developers and coding agents refine graphs through the *Agint CLI*, while non-technical users leverage *Agint Flow GUI* with visual editing, conversational refinement, and debugging to promote prototype agentic workflows to production code. This continuous co-creation model lets teams prototype quickly, refine seamlessly, and deploy reliably, bridging natural language, compiler methods, and developer tooling to enable a new generation of composable, team-centric coding agents at scale.

1 Introduction

Large Language Models (LLMs) have enabled coding agents that translate natural language into executable code [4, 11]. However, in real-world software engineering, there are still issues to address: syntactic errors, hallucinated functionality, and training-distribution quirks often require significant human correction. Models struggle with project-specific references in limited context, degrade with very long contexts, and either run slowly (large models) or prove unreliable (small models), especially for domain-specific tasks and structured outputs [5, 9].

These issues intensify in multi-agent and multi-developer settings. Concurrent edits introduce contention and cascading errors, as agents lack reliable coordination [15, 6]. Unstructured generation is fast but fragile, while structured generation is more controllable but slower, particularly for specialized models without a fast inference provider [14]. Multi-model delegation through standards such as Multi-Client Proxy (MCP) adds token overhead, latency, variability, and dependency on opaque third-party implementations.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Deep Learning For Code in the Agentic Era.

Software engineering rarely involves code alone: it requires coordination with datasets, APIs, and external connectors. Current agents cannot robustly integrate across these resources while staying efficient [13, 17]. Even when producing step-by-step plans, execution is sequential and error-prone, with limited ability to parallelize or adapt to runtime values. Long-reasoning models exacerbate this by delaying early parallelism. Auxiliary tasks such as browsing, querying, or data transformation interrupt workflows and create dependencies that are difficult to track. In practice, code generation is inseparable from *data organization* and *workflow orchestration* [12].

While traditional programming elevated data structures and algorithms as the foundation of computation, the agentic era demands *semantic data organization* and *agentic workflow orchestration*. **Agint** addresses this by compiling natural-language intent into typed, effect-aware Directed Acyclic Graphs (DAGs) that unify code generation, data handling, workflow orchestration, and agent coordination. By grounding interactions in graphs, Agint ensures concurrency safety and composability across code, data, and tools, turning code generation from a long-running for-loop of fragile one-shot predictions into a structured process of agentic co-creation.

The graph form enables *recursive concurrent task decomposition*: complex problems are partitioned into independent subgraphs that can be solved in parallel, refined incrementally, and recomposed without global bottlenecks [2, 19]. Because Agint treats workflow design as a compilation problem, heavy models can be used for *ahead-of-time agentic compilation*, laying out the workflow structure, while lightweight, low-latency models execute and refine individual nodes [20].

After the workflow skeleton is created, it undergoes two concurrent processes: *refinement* and *resolution*. Refinement enriches each node with unstructured contextual information—examples, constraints, documentation, or domain knowledge—that guides downstream generation without altering the type system [10]. Resolution, by contrast, upgrades the node types in a structured manner: a natural-language node is successively promoted into a function signature, then into a formal specification, and finally into an executable code implementation [3]. Importantly, Agint does not require waiting until full resolution to run a workflow: nodes can be executed in simulated or mock modes at intermediate stages, enabling early testing, speculative execution, and validation before compilation. Together, refinement and resolution preserve the graph's structure while progressively attaching typed bindings and contextual detail, making execution modular, reproducible, and tightly integrated across the workflow.

Agint's DAGs localize changes to graph nodes, allowing small, fast models to solve subproblems in parallel, escalating to larger models only when needed [1, 21]. Each node inherits context from its ancestry, minimizing global overhead. Classical compiler and runtime techniques—JIT-style optimization, speculative graph evaluation—further improve efficiency and reliability. This reduces context overhead, increases throughput, accuracy, and enables transparent, reproducible workflows [7].

Agint thus bridges flow-based programming with agentic autonomy, providing a foundation for scalable, collaborative coding agents. In this paper, we introduce Agint's compiler, interpreter, and runtime, and show how they enable reliable, scalable, and user-friendly workflows for code generation and data transformation.

2 System Overview

Agint implements a multi-stage graph compilation pipeline that transforms natural language specifications into executable code through a hierarchy of typed intermediate representations. Unlike traditional code generation approaches that operate on isolated functions, our system compiles entire workflows as directed acyclic graphs (DAGs), where nodes progress through six distinct type floors: `TEXT` $\rightarrow$ `TYPED` $\rightarrow$ `SPEC` $\rightarrow$ `STUB` $\rightarrow$ `SHIM` $\rightarrow$ `PURE`.

Each type floor represents a computational abstraction with well-defined semantics:

- **TEXT**: Natural language descriptions with implicit data flow, and task dispatch if the underlying execution LLM supports it
- **TYPED**: Nodes acquire explicit type signatures (`PrimitiveType` system constraining to `str`, `int`, `float`, `bool` and their lists)
- **SPEC**: Formal specifications with pre/post conditions and behavioral constraints

- **STUB**: Function signatures with stub implementations
- **SHIM**: Hybrid execution nodes containing both deterministic code and AI-synthesized virtual functions
- **PURE**: Fully resolved executable code with no AI dependencies

The key innovation lies in the executability of intermediate representations. A `TYPED` node can execute through prompt chaining, while a `SHIM` node employs hybrid execution—deterministic code paths invoke AI-generated virtual functions (`VIRTUALSTUB`, `VIRTUALSHIM`, `VIRTUALPURE`) on demand. This enables incremental development where partially-specified workflows remain runnable throughout the compilation process.

2.1 Compilation Architecture

The compilation pipeline employs three core mechanisms:

1. Type-Directed Resolution: Each node maintains a `RESOLUTION_STATE` (`UNRESOLVED`, `IN_PROGRESS`, `PARTIALLY_RESOLVED`, `FULLY_RESOLVED`) that guides the compiler's strategy. Resolution preserves semantic equivalence while upgrading representations—a `TYPED` node's behavioral specification becomes explicit pre/post conditions in `SPEC`, which translate to function signatures in `STUB`.

2. Locality-Preserving Transformation: The DAG structure localizes compilation context to node neighborhoods. When resolving a node from `SPEC` to `STUB`, the compiler considers only immediate dependencies and dependents, not the entire graph. This enables parallel compilation where independent subgraphs resolve concurrently without global synchronization.

3. Fallback Synthesis: When direct compilation fails (e.g., ambiguous specifications, complex domain logic), the system employs three fallback strategies:

- *Decomposition*: Complex nodes split into multiple simpler nodes
- *Virtual Functions*: Unresolvable logic becomes `VIRTUALSHIM` nodes for runtime synthesis
- *Deferred Compilation*: Nodes marked for resolution in subsequent passes

2.2 Runtime Execution Model

The runtime implements a hybrid execution strategy that seamlessly blends compiled code with AI-powered synthesis. Three execution modes support different performance/flexibility tradeoffs:

Prefine Mode: Runtime optimizes the nodes' function before execution while it's waiting for input from currently running nodes. This mode optimizes runtime codegen quality when code is later dynamically generated.

Dynamic Mode: Nodes execute with just-in-time synthesis. When encountering a `VIRTUALSHIM`, the runtime generates implementations using upstream execution context. This enables adaptive behavior where node implementations specialize based on actual data flow.

Predict Mode: Speculative execution runs multiple node implementations in parallel. The runtime predicts likely execution paths and pre-generates likely function input arguments and pre-emptively executes the function, hiding synthesis, and execution latency through prediction.

2.3 System Components

The architecture comprises specialized subsystems interconnected through well-defined interfaces:

Flyte provides unified LLM orchestration with hierarchical structured generation. It implements prompt registries, multi-provider routing, and automatic failover. The integration with Hydantic inference enables decomposition of complex outputs into parallel subtasks, reducing generation latency by 3-10× for large structured outputs with multiple independent fields.

Dagify serves as the graph compiler, implementing the type floor progression through a series of specialized resolvers. Each resolver (`PlainToTypedResolver`, `TypedToSpecResolver`, etc.)

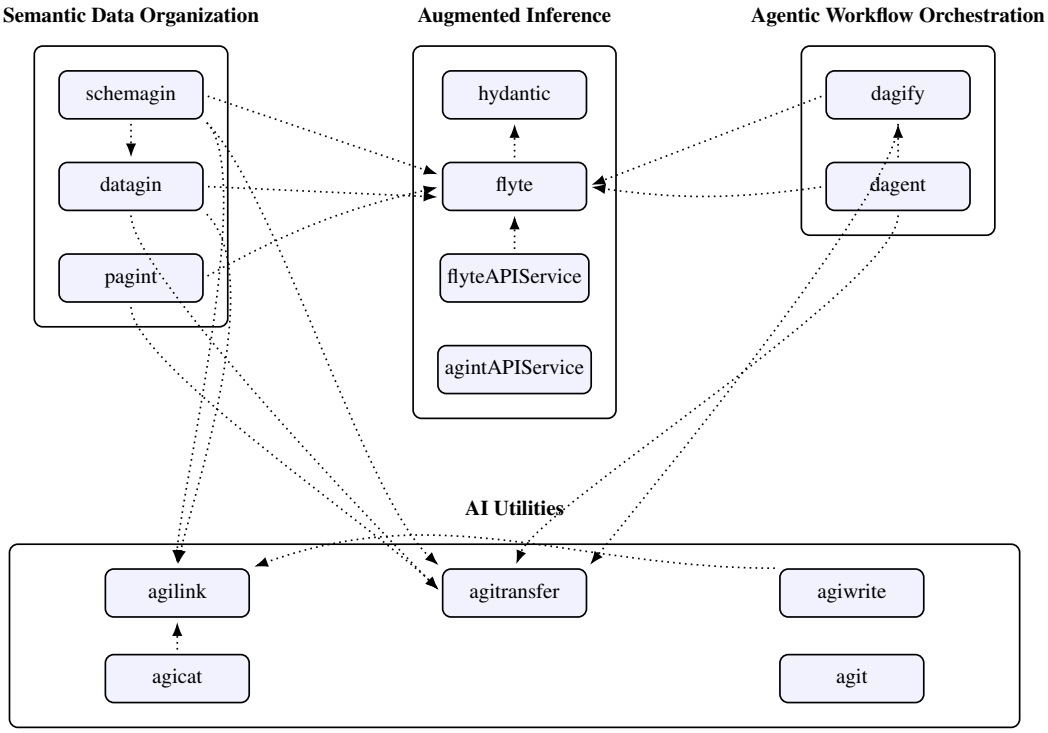


Figure 1: Architecture Overview

handles one transition in the type hierarchy. The compilation process supports incremental refinement where nodes can be individually upgraded without full recompilation.

Dagent implements the hybrid runtime with effect-aware execution. It tracks all side effects (filesystem, network, database operations) through an effect monad, enabling safe rollback and reproducible execution despite non-deterministic AI components. The runtime partitions DAGs into independent subgraphs that execute concurrently on separate workers. Additionally, Dagent is capable of runtime tool use of Dagify, Schemagin, Datagin, and MCP.

Schemagin/Datagin: Schemagin generates typed schemas from natural language, while Datagin handles data transformation and synthesis. Both integrate through Agilink (unified data connectivity) and Agitransfer (versioned artifact management), providing a complete prompt-to-database pipeline that mirrors the code compilation process.

2.4 High-Level Architecture and Data Flow

Data Flow: *Dagify* generates and evolves DAGs through *Flyte* and produces versioned YAML/JSON artifacts. *Dagent* executes DAGs (or interprets them on the fly), invoking LLMs only through Flyte, and interacting with *Agilink* for remote database connectivity and *Agitransfer* for version control. *Schemagin* primarily writes schemas (SQL/JSON/YAML/DBML) that *Agilink* can read; *Datagin consumes schemas from Schemagin* (when provided) and performs *ingest/synthesis/transform*, reading/writing data through *Agilink*. All components share consistent CLI patterns and Pydantic-like models registered with Flyte which are dynamically converted to BAML.

2.5 How the System Works

2.5.1 Dagify - Compose/Refine/Resolve/Compile

Users or agents provide an NL specification; Dagify composes a *Plain* DAG, refines nodes with context, resolves them to *Typed* and *Spec* nodes, and optionally compiles to *Code* DAGs or target-specific builds (e.g., CrewAI, LangChain, WDL, BAML). All LLM calls (multi-shot, with fallback) are mediated by Flyte, with structured outputs validated against registered models.

2.5.2 Dagent - Execute/Interpret

Dagent executes a precompiled DAG, interprets and runs dynamically, or compiles-then-runs. The runtime supports speculative execution and effect-aware transactions/rollback, and partitions independent subgraphs for parallelism.

2.5.3 Schemagin & Datagin

Schemagin generates schemas and visualizations; Datagin ingests unstructured inputs, synthesizes test data, or transforms datasets against provided schemas. Both route I/O through Agilink (DBs) and Agitransfer (files) as needed.

2.6 Flyte: Key Benefit and Why It Is Fast

Centralized, async, multi-provider structured generation. Flyte is the single LLM gateway: it manages prompt/structured-output registries, routes across providers, and executes *asynchronously by default*, enabling high concurrency under rate limits. Crucially, Flyte integrates **Hydantic** (below) so complex structured outputs are generated via *hierarchical, parallel subcalls* with focused context, which improves latency and reduces required context with useful accuracy. Together, multi-provider routing + async orchestration + Hydantic's parallel structured filling are why Flyte-backed operations are robust and fast in practice.

2.7 Hydantic in Context (What It Does)

Hydantic, extends Pydantic with *hierarchical decomposition* of nested models into levels (simple fields, Pydantic objects, nested Hydantic models). Flyte uses this to *parallelize* field- or sub-model generation with *focused prompts* per branch, decreasing per-call context and improving fill accuracy. The result is *concurrent, progressive* population of complex outputs (often yielding substantial speedups (relative to a more expensive model) for models with many independent fields). Hydantic is automatically detected by Flyte's frameworks and used when a Hydantic model is registered as the structured output. (The name Hydantic is a portmanteau of Huygens + Pydantic, because the inference frontier propagates through the Pydantic model via Huygens principle.)

2.8 What Schemagin and Datagin Actually Do (and Their Data Flows)

Schemagin converts NL descriptions into database schemas, refines existing schemas, and renders multiple formats (SQL, JSON, YAML, DBML) with visualization support (GraphViz/D2/ASCII). Its outputs are *schema artifacts* that *Agicat* and *Agiwrite* can read and write, which *Datagin* can consume as *schema input*.

Datagin supports three roles: *ingest* (extract structure from un/semistructured sources), *synthesis* (generate schema-conformant test data), and *transform* (convert between formats/schemas). Datagin *optionally* takes Schemagin's schema as input and performs reads/writes through *Agilink* and *Agiwrite* and spawns fully formed databases. Both Schemagin and Datagin register prompts/structured outputs with *Flyte* and adhere to the shared CLI/async generation patterns.

3 Agint CLI

Agint implements a Unix-style toolchain where each CLI tool performs a specific function in the compilation and execution pipeline. The tools communicate through structured text formats (YAML/JSON/DBML) and a unified addressing system (`agilink://`), enabling flexible composition while maintaining type safety and reproducibility.

3.1 Workflow Orchestration Tools

dagify serves as the DAG compiler, transforming natural language specifications into progressively refined workflow graphs. It implements four primary operations:

- `compose`: Generates initial DAGs from natural language at a specified type floor

- `refine`: Adds contextual information to nodes without changing types
- `resolve`: Upgrades DAGs to higher type floors through AI-assisted compilation
- `compile`: Produces executable artifacts for target platforms (WDL, CrewAI, LangChain, BAML)

dagent provides the runtime environment for DAG execution. It supports multiple execution strategies:

- `validate`: Performs static analysis and type checking
- `optimize`: Tunes prompts and parameters against test datasets
- `execute`: Runs DAGs with configurable JIT modes (prefine, dynamic, predict)
- `interpret`: Generates and executes workflows directly from natural language
- `synthesize`: Combines plan generation, compilation, and execution in one pass

Execution is constrained by explicit tool whitelists (`-tools`), ensuring reproducibility and preventing unintended side effects.

3.2 Data Management Tools

schemagin generates and manipulates database schemas from natural language. It provides schema authoring (`compose`), iterative refinement (`refine`), and multi-format visualization (`visualize`) supporting SQL, JSON, YAML, and DBML outputs with ASCII, GraphViz, and D2 rendering.

datagin handles data transformation workflows through three core operations:

- `ingest`: Extracts structured data from unstructured sources
- `synthesize`: Generates realistic test data conforming to schemas
- `transform`: Converts data between schemas with rule generation

agicat and **agiwrite** provide bidirectional synchronization between schema definitions and live databases, enabling round-trip engineering between design and deployment.

pagint offers row-level data operations, generating individual records or augmenting existing datasets with missing fields. It complements datagin's batch operations with fine-grained, internet-aware data synthesis.

3.3 Integration Architecture

The `agilink://` URI system provides a typed addressing layer that abstracts over storage backends. Components reference schemas, datasets, and transformations uniformly regardless of whether they reside in files, databases, or memory. This abstraction enables:

- Type-safe data flow between tools
- Transparent caching and versioning
- Distributed execution without code changes
- Reproducible artifact references across environments

All tools share common CLI conventions (`-seed` for reproducibility, `-intelligence` for model selection, `-format` for output control) and integrate through Flyte for LLM operations, ensuring consistent behavior across the toolchain.

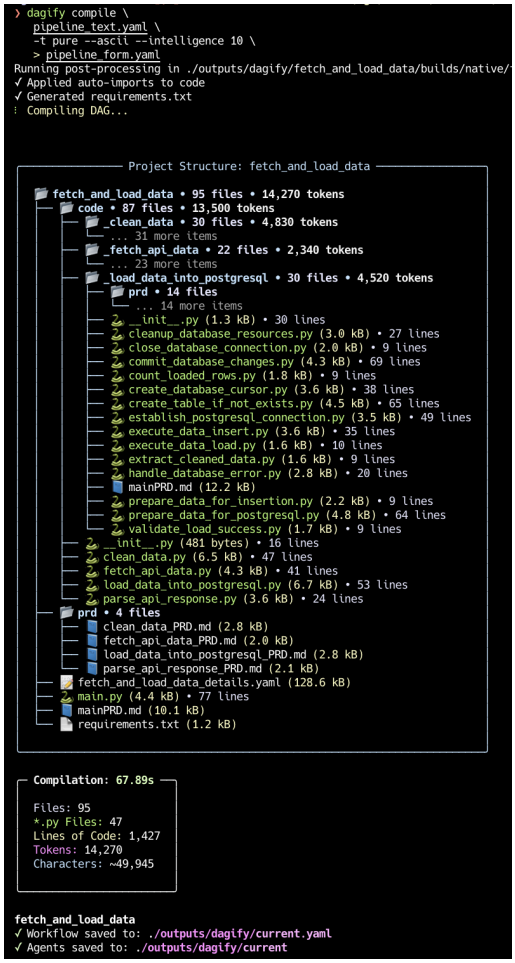
4 Usage Examples

4.1 Create and Compile an ETL Data Processing Workflow

```
dagify compose --type plain_text -f yaml --ascii \
```

```
  "Fetch data from API → clean data → load into PostgreSQL" \
  > pipeline_text.yaml

dagify compile pipeline_text.yaml -t pure --ascii --intelligence 10 \
   > pipeline_form.yaml

# Step 2: Execute the compiled workflow with dagent
dagent execute pipeline_form.yaml --tools=python,psql
```

(a) Output of `dagify compose` showing the generated DAG structure.



(b) Output of `dagify compile` showing the code DAG file structure.

Figure 2: Comparison of outputs from `dagify compose` and `dagify compile`.

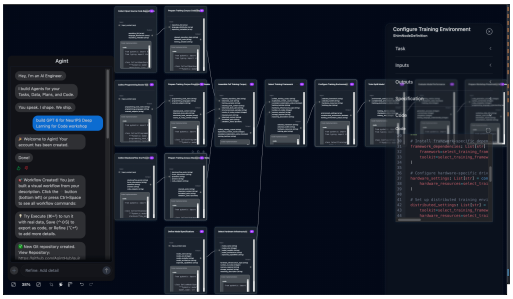
4.2 Schema and Data Management

```
# Generate a db schema for a blog with posts and comments
schemagin compose --format=json -f dbml \
    "Blog posts & comments feel

# Apply schema to database and generate test data
agiwrite schema --target-dbfer=duckdb://orders.db < schema.yaml
datagin synthesize "1000 orders" schema.yaml --rows=1000 \
  --output-agilink agilink://orders
```

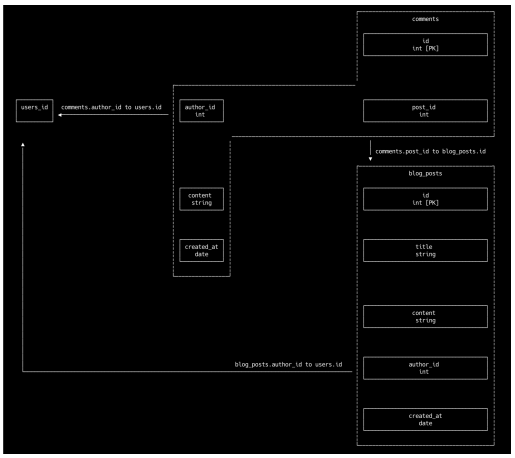
```
# Transform data with anonymization
datagin transform "Remove PII" agilink://orders agilink://orders_clean
```

4.3 End-to-End Example: Analytics Pipeline

```
# 1. Design schema
schemagin compose "Sales analytics schema" -f yaml > sales.yaml

# 2. Create workflow
dagify compose -t spec "ingest CSV -> validate -> aggregate -> report" \
  > pipeline.yaml

# 3. Resolve and compile
dagify resolve pipeline.yaml --type-floor=code > pipeline.code.yaml
dagify compile pipeline.code.yaml --build-target=wdl > pipeline.wdl

# 4. Execute with monitoring
dagent validate pipeline.wdl
dagent execute pipeline.wdl --jit=dynamic --tools=python,duckdb
```



(a) Agint Flow GUI.



(b) Screenshot of Automated Schema Generation via Agint CLI

5 Limitations and Societal Impacts

5.1 Limitations

While Agint demonstrates significant advances in graph-based code generation, several limitations warrant discussion:

Model Dependency: The system's effectiveness depends heavily on the quality and availability of underlying LLMs. Performance degrades when models lack domain-specific knowledge or when rate limits restrict concurrent execution. The multi-provider routing in Flyte mitigates but does not eliminate this dependency.

Scalability Constraints: Although the DAG structure enables parallel execution, very large workflows (thousands of nodes) may encounter memory limitations during compilation and execution. The current implementation has been tested primarily on workflows with hundreds of nodes.

Type System Limitations: The `PrimitiveType` system constrains data to basic types (`str`, `int`, `float`, `bool`) and their lists. Complex data structures require serialization, potentially limiting expressiveness for certain domains.

Evaluation Gaps: While we demonstrate the system's capabilities through examples and case studies, comprehensive evaluation against established benchmarks remains future work. Existing benchmarks like SWE-bench [8] for repository-level issue resolution, ML-Bench [16] for machine learning workflows, and Commit0 [22] for library generation would provide valuable comparative assessment of Agint's graph-based approach versus traditional methods.

5.2 Societal Impacts

Agint could democratize programming by enabling domain experts to specify complex workflows in natural language, potentially accelerating software development cycles through parallel generation and typed guarantees. The effect-aware execution model promotes reproducible AI-assisted development. However, concerns include potential displacement of entry-level programming jobs and organizational dependency on LLM providers.

6 Conclusion and Future Work

6.1 Conclusion

We presented Agint, an agentic graph compiler that transforms natural language specifications into executable code through hierarchical type floor-driven generation and hybrid execution. By treating code generation as a graph compilation problem, Agint addresses fundamental limitations of current LLM-based coding assistants: context management, reliability, and composability.

Our key contributions include:

- A six-tier type floor system (`TEXT` $\rightarrow$ `TYPED` $\rightarrow$ `SPEC` $\rightarrow$ `STUB` $\rightarrow$ `SHIM` $\rightarrow$ `PURE`) that enables incremental refinement and execution at any resolution level
- Locality-preserving DAG compilation that enables parallel generation and reduces context requirements
- A hybrid runtime supporting prefine, dynamic, and predict execution modes with effect-aware rollback
- A composable Unix-style toolchain (dagify, dagent, schemagin, datagin) unified through the `agilink://` addressing system

The system demonstrates that compiler techniques—type systems, intermediate representations, and optimization passes—can effectively structure AI-powered code generation, making it more reliable and scalable than monolithic approaches.

6.2 Future Work

Extended Type Systems: Future versions will support richer type systems including algebraic data types, generics, and effect types to enable more precise specifications and better integration with strongly-typed languages.

Formal Verification Integration: Formal verification tools at the `SPEC` floor may provide stronger correctness guarantees. We envision automatic generation of proof obligations from specifications.

Learning from Execution: Feedback loops where execution results will improve future compilations may create self-improving systems. Failed executions would generate training data for model fine-tuning.

Benchmark Suite: We will evaluate Agint on established benchmarks including SWE-bench [8] [18], ML-Bench [16], and Commit0 [22] to quantify the benefits of the graph-based compilation approach over monolithic generation methods.

Agint represents a step toward treating AI code generation as a structured compilation problem rather than a text generation task. By grounding natural language in typed graphs and providing incremental refinement mechanisms, we enable a human-AI collaboration at scale in software development. As demand and utilization of LLMs increase, improve, the principles demonstrated here—locality, typing, and effect awareness—will become increasingly important for building reliable, scalable software engineering agents.

Acknowledgments and Disclosure of Funding

We thank the reviewers and organizers of the DL4C Workshop at NeurIPS 2025 for their constructive feedback and valuable discussions that helped improve this work.

We would also like to sincerely thank **Ahmed El-Kishky** for his encouragement and for believing in us throughout this project.

This work was supported solely by internal research efforts at Agint and received no external financial support.

References

[1] Daman Arora et al. Masai: Modular architecture for software-engineering ai agents. 2024. Available at arXiv:2406.11638.

[2] Maciej Besta et al. Graph of thoughts: Solving elaborate problems with large language models. 2024. Available at arXiv:2305.16582.

[3] Jingchang Chen et al. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. 2024. Available at arXiv:2405.20092.

[4] Mark Chen et al. Evaluating large language models trained on code. 2021. Available at arXiv:2107.03374.

[5] Alex Gu et al. Challenges and paths towards ai for software engineering. 2025. Position paper, available at arXiv:2503.22625.

[6] Shuyue Hong et al. Metagpt: Meta programming for a multi-agent collaborative framework. 2023. Available at arXiv:2308.00352.

[7] Yaojie Hu et al. Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks. 2025. Available at arXiv:2501.17167.

[8] Carlos E. Jimenez et al. Swe-bench: Can language models resolve real-world github issues? 2023. Available at arXiv:2310.06770.

[9] Darren Key et al. Toward trustworthy neural program synthesis. 2022. Available at arXiv:2210.00848.

[10] Hung Le et al. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. 2024. Available at arXiv:2310.08992.

[11] Yujia Li et al. Competition-level code generation with alphacode. 2022. Available at arXiv:2203.07814.

[12] Ziming Li et al. Autokaggle: A multi-agent framework for autonomous data science competitions. 2024. Available at arXiv:2410.20424.

[13] Junwei Liu et al. Large language model-based agents for software engineering: A survey. 2024. Available at arXiv:2409.02977.

[14] Niels Mündler et al. Type-constrained code generation with language models. 2025. Workshop paper, DL4C @ ICLR 2025, available at arXiv:2504.09246.

[15] Chen Qian et al. Chatdev: Communicative agents for software development. 2024. Available at arXiv:2307.07924.

[16] Xiangru Tang et al. Ml-bench: Evaluating large language models and agents for machine learning tasks on repository-level code. 2023. Available at arXiv:2311.09835.

[17] Yanlin Wang et al. Agents in software engineering: Survey, landscape, and vision. 2024. Available at arXiv:2409.09030.

[18] John Yang et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? 2024. Available at arXiv:2410.03859.

[19] Shunyu Yao et al. Tree of thoughts: Deliberate problem solving with large language models. 2023. Available at arXiv:2305.10601.

[20] Janis Zenkner et al. Shedding light on task decomposition in program synthesis: The driving force of the synthesizer model. 2025. Accepted at ICLR 2025 Workshop, available at arXiv:2503.08738.

[21] Kexun Zhang et al. Diversity empowers intelligence: Integrating expertise of software engineering agents. 2024. Available at arXiv:2408.07060.

[22] Wenting Zhao et al. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025. Available at OpenReview.

7 Technical Appendices and Supplementary Material

7.1 Interactive Demo and Documentation

We provide comprehensive supplementary materials at the following locations:

- **Interactive Web Demo**: `https://flow.AgintAI.com` - A web-based interface demonstrating Agint Flow GUI with visual DAG editing, workflow execution, and real-time compilation from natural language specifications.
- **API Documentation**: `https://api.AgintAI.com` - Complete API reference for programmatic access to Agint's compilation and execution services, including REST endpoints, authentication, and usage examples.

7.2 Example Generated Code

An example of Agint-generated workflow code for training a large language model workflow can be found at:

- `https://github.com/AgintHub/nifty-wilson/blob/agint/outputs/dagify/train_large_language_model/main.py`

This demonstrates the compiled output from natural language specification to executable Python code, showcasing the type floor progression and modular decomposition of complex workflows into manageable components.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction accurately present Agint as an agentic graph compiler that transforms natural language into executable code through type floors (TEXT→TYPED→SPEC→STUB→SHIM→PURE), hybrid execution modes (pre-fine/dynamic/predict), and a Unix-style toolchain. All claims are substantiated in the technical sections.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 6.1 explicitly discusses limitations including model dependency (performance depends on LLM quality/availability), scalability constraints (tested primarily on hundreds of nodes, not thousands), and type system limitations (constrained to primitive types and their lists).

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: This is a systems paper presenting a software architecture and implementation, not a theoretical paper with formal proofs.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides detailed system architecture, type floor definitions, execution modes, and concrete CLI examples showing how to use each component (dagify, dagent, schemagin, datagin). While no quantitative experiments are presented, the system design and workflow examples are described with sufficient detail for implementation.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: The paper describes a system architecture and implementation approach but does not provide open-source code. The supplementary material at flow.AgintAI.com provides additional documentation and demos rather than reproducible code.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [NA]

   Justification: The paper does not include experiments requiring training or testing. It presents a system architecture and compilation framework.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [NA]

   Justification: The paper does not include experiments with statistical results. It focuses on system design and architecture.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [NA]

   Justification: The paper does not include computational experiments. It describes system architecture and tools.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research presents a software system for code generation without involving human subjects, sensitive data, or ethically concerning applications.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Section 6.2 (Societal Impacts) discusses both positive impacts (democratizing programming for domain experts, accelerating development cycles) and negative impacts (potential displacement of entry-level programming jobs, organizational dependency on LLM providers).

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: The paper does not release pretrained models, image generators, or scraped datasets. While the system includes effect-aware execution and rollback capabilities for safe code execution, these are operational features rather than release safeguards for high-risk assets.

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: The system uses commercial LLM APIs. No external code or datasets requiring special licensing are incorporated.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [No]

Justification: The paper introduces a new system (Agint) with novel components as a demo (Flyte, Dagify, Dagent, etc.), it does not release these as downloadable assets. The paper provides architectural documentation and usage examples, but the actual software implementation is not made publicly available. The paper is a Technical and Demo paper.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: The research does not involve crowdsourcing or human subjects. It presents a software system architecture.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The research does not involve human subjects and therefore does not require IRB approval.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The paper extensively describes LLM usage as a core component. Section 2.4 details how Flyte orchestrates multiple LLM providers with automatic failover and hierarchical structured generation through Hydantic. LLMs are fundamental to transforming natural language to code across all type floors.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.