Polymer: Development Workflows as Software

Citation for the original published paper (version of record):
Parthasarathy, D., Yu, Y., Barr, E. (2025). Polymer: Development Workflows as Software. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering: 535-539. http://dx.doi.org/10.1145/3696630.3728494

Polymer: Development Workflows as Software

Dhasarathy Parthasarathy
Volvo Group
Gothenburg, Sweden
dhasarathy.parthasarathy@volvo.com

Yinan Yu
Chalmers University of Technology
Gothenburg, Sweden
yinan@chalmers.se

Earl T. Barr
University College London
London, UK
e.barr@ucl.ac.uk

ABSTRACT

Software development builds digital tools to automate processes, yet its initial phases, up to deployment, remain largely manual. There are two reasons: Development tasks are often under-specified and transitions between tasks usually require a translator. These reasons are mutually reinforcing: it makes little sense to specify tasks when you cannot connect them and writing a translator requires a specification. LLMs change this cost equation: they can handle under-specified systems and they excel at translation. Thus, they can act as skeleton keys that unlock the automation of tasks and transitions that were previously too expensive to automatically interlink. We introduce a recipe for writing development workflows as software (`polymer`) to further automate the initial phases of development. We show how adopting `polymer` at Volvo, a large automotive manufacturer, to automate testing saved 2–3 FTEs at the cost of two months to develop and deploy. We close with open challenges when `polymer`izing development workflows.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**; **Software development process management**.

KEYWORDS

Software development automation, Large Language Models

ACM Reference Format:
Dhasarathy Parthasarathy, Yinan Yu, and Earl T. Barr. 2025. Polymer: Development Workflows as Software. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728494

1 INTRODUCTION

Software Engineering (SE) in large organizations is a complex undertaking [8]. All SE organizations aim to operate this complex process at high cadence while delivering quality software products. Achieving this goal takes tremendous effort because it requires tackling SE process complexity as a whole. For instance, in a large automotive company like Volvo, developing a portfolio of 3-5 vehicle platforms involves hundreds of engineers developing and integrating thousands of SW components with thousands of mechatronic and physical components. The combinatorial explosion of interactions means that management and engineering teams can lose track of each other's roles and responsibilities. The resulting *awareness gap* increases waste, slows development, and reduces quality.

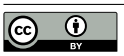
Automation manages complexity better: Consider a typical SE process with phases such as specification, implementation, verification/validation, deployment, and operation. Interestingly, the tail-end phases already employ a powerful approach for handling complexity — treating entire phases like deployment or operations as software problems [3] and automating workflows by writing them as software. Thanks to the widespread adoption of DevOps, aided by a deep tech stack, engineers code complex workflows that continuously build, test, deploy, and monitor SW products, some at global scale catering to millions of users. How can SW-defined DevOps workflows handle high complexity? We offer a few explanations:

- SW-defined workflows are more formal, making them clearer, bridging the awareness gap between stakeholders.
- Parts of SW-defined workflows can be executed, obviating manual work and providing faster feedback with less coordination.
- Workflow instrumentation and logging permits close observation and measurement, which enables organizations to control complex SW processes with leaner management.
- SW-defined workflows can be simulated, enabling continuous refinement to be more reliable, observable, and faster.

Given these benefits and the success of SW-defined workflows in handling DevOps complexity, we ask: "Can we left-shift the idea of SW-defined workflows and replicate this success in earlier SE phases?". The `polymer` vision centers on answering this question.

The `polymer` Challenge: We model an SE workflow as a graph where nodes represent tasks and edges represent transitions, or information flows, between tasks. Within this structure, the level of specification varies. Product tasks, which produce tangible outcomes like software or documentation, tend to be better specified. Control tasks such as coordination or hand-off tend to be under-specified. Transitions are usually as well specified as the tasks they interconnect. As workflow complexity and awareness gaps grow, so does the level of under-specification. Applying `polymer` involves automating as many tasks and transitions as possible, so it requires addressing under-specification directly. For tasks, automation entails formalizing inputs and outputs, and making it executable, which is expensive. Likewise, automating transitions requires translators capable of parsing, interpreting, and processing the information flows, which is equally expensive. These costs can trigger a vicious *under-specification cycle* where high costs and awareness gaps lead to vague specifications. This hinders automation and, with feature growth, increases complexity. The increased workload leaves no room for improving specifications, perpetuating the cycle. Often, extra human intervention is required to keep this cycle from spiraling.

How did DevOps break out of this cycle and achieve mature SW-defined workflows? We argue that this success is primarily due



FSE Companion '25, June 23–28, 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728494

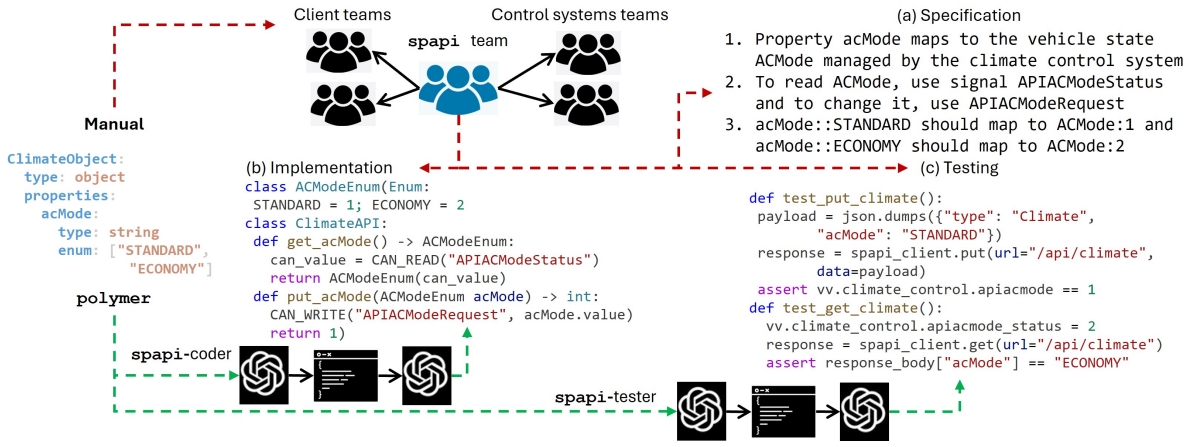


Figure 1: `spapi` is representative of a complex SE process involving continuous coordination during specification, implementation, and testing. Applying `polymer`, this workflow becomes SW-defined and automatic, reducing development complexity and saves thousands of person hours. Note: examples have been sanitized to protect proprietary information.

to the standardization of tail-end SE tasks. Diverse applications, from media streaming to vehicle embedded systems, have implicitly joined forces to standardize tasks such as CI/CD, regression testing, infrastructure provisioning, and observability. This standardization creates high demand and reuse potential, justifying years of collective effort to progressively automate tail-end tasks and transitions.

However, if we shift left from the tail-end into the earlier SE phases, we risk slipping into the long tail of workflow diversity, where some workflows are so domain-specific as to be unique. For such cases, the cost of conventional automation remains prohibitive. Why, then, do we advocate adopting `polymer` in early phases of development? We do so because of the advent of a crucial tipping factor — Large Language Models (LLMs). An LLM's general task solving ability dramatically changes the economics of handling under-specification and automating task chains, with less concern about standardization, formalization, or executability.

LLMs as "skeleton keys" unlock automation opportunities:

(1) LLMs are task-agnostic [5, 6, 21], and can be adapted to solve diverse tasks using lean prompting. This is often cheaper than implementing every task as code.
(2) LLMs handle semi-structured and semi-formal inputs, reducing, and sometimes eliminating, data wrangling [13, 19].
(3) LLM outputs can be structured and adhere to syntactic constraints [2, 20, 22], allowing them to be composed with conventional, deterministic functions in pipelines.
(4) LLM have a flexible front-end, supporting natural language, programmatic [1, 15], and multi-modal interactions, enabling diverse SE stakeholders to solve tasks with them.
(5) LLMs can serve as a flexible back-end, addressing SW tasks, data tasks, and even some HW tasks [4, 18].

Based on these observations, we claim that *LLMs are skeleton keys for development workflow automation*. That is, LLMs enable automation with far less effort than conventional methods, making it feasible to automate, *aka polymerize*, task sequences that were previously prohibitively expensive to automate. Building upon this claim, we propose a simple recipe for applying `polymer` and reducing the cost of handling SE complexity: Take a development workflow and (a) identify tasks and information flows suitable for automation, (b) write these tasks and flows as *code*, invoking LLMs where needed and using conventional automation otherwise. Taking concrete steps towards `polymer`, we contribute the following:

(1) We present a real-world case of test workflow as software. Developed in two months, `spapi-tester` is a `polymer` process that automates testing `spapi`— a vehicle web server produced by Volvo— and saved the effort equivalent to 2-3 FTEs.
(2) We discuss a real-world case of implementation workflow as software. Being piloted at Volvo, `spapi-coder` aims to `polymer`-ize `spapi` implementation.
(3) We lay out a research agenda for `polymer` enabled by LLMs as skeleton keys for automation.

2 A MOTIVATING EXAMPLE

For our discussion, we use the development workflow of `spapi` (Figure 1) as a running example. It represents our target application domain — a complex workflow marked by under-specification, awareness gaps, and continuous coordination. A production web server deployed in Volvo trucks, `spapi` offers RESTful APIs allowing mobile applications to access and modify vehicle state. For example, the `/speed` endpoint is used to read the vehicle's speed, while `/climate` adjusts the cabin climate settings.

Specification is a coordination vortex. While specification is always a challenging aspect of development, workflows with multiple stakeholders can demand overwhelming coordination. With `spapi`, it often takes several discussions before requirements engineers even recognize the need for an endpoint like `/climate` with a property `acMode` (see Figure 1). Using their domain knowledge, they link this endpoint to the climate control system. However, the organizational and disciplinary separation between `spapi` and climate engineers creates awareness gaps, driving up the cost of maintaining robust specifications. Incomplete or outdated vehicle state documentation forces manual coordination to match `ACState` with `acMode`. Moreover, because vehicle states are accessed via Controller Area

Network (CAN) signals, spapi architects must also identify or define the associated read and write signals (APIACModeStatus and APIACModeRqst), requiring further coordination with the signal definition team. After extensive coordination, the mapping between a property, signal, and state is recorded using natural language text (Figure 1(a)). Since spapi comprises hundreds of APIs and thousands of attributes, Volvo spends thousands of hours on specification. Even so, clarity and stability in specification is hard to achieve.

Implementation can fall into the coordination vortex. Given clear specifications, translating them into a working implementation can be straightforward in processes like spapi. For an endpoint like /climate, developers align data types between the property acMode, and state ACState, before using CAN signals APIACMode* to read and write vehicle state (Figure 1(b)). However, due to constant changes in property, signal, and state definitions, specifications inevitably degrade, forcing developers to coordinate with stakeholders, adding hundreds of hours of overhead.

Testing is also claimed by the vortex. Ambiguity and frequent specification changes inevitably affect testing. For spapi, testers manually write and maintain test cases to ensure that endpoints comply with the specified interface and correctly map API properties to vehicle states. For example, they write a test case for a PUT method that sends acMode:STANDARD and checks if ACState changes to 1 (Figure 1(c)). However, since they rely upon the same set of degrading specifications, testers are forced to spend hundreds of additional hours coordinating with requirements engineers, developers, and close stakeholders to resolve differences.

Effective coordination and precise specifications are essential in SE, yet real-world workflows like spapi continually struggle with deficiencies in both. The usual response is to introduce more management, restructure organizations, and enforce process discipline — measures that are expensive and offer no guarantees. Conventional automation might help, but the unique nature of these workflows and the need for extensive change make it costly and unlikely to be prioritized. How can we break out of this stalemate and improve the process? Volvo is piloting polymer using LLMs as skeleton keys.

3 TEST WORKFLOW AS CODE

A typical test workflow involves translating specifications into test cases and executing them to assess quality. In real-world settings, as seen in the spapi case, it needs to address the complexity arising out of under-specification and awareness gaps between multiple stakeholders. Left shifting by applying polymer, Volvo addresses these issues by developing spapi-tester[1] (Listing 1), a SW-defined workflow that automates the test process. Testing spapi starts with reading specifications (e.g., mapping acMode:STANDARD to ACState:1), followed by determining mocks and test inputs and integrating test cases into a regression suite — a process that takes 2-3 FTEs. Rather than building a costly semantic parser [14], spapi-tester uses an LLM as a translator via the dspy framework [15]. Invoking LLM calls through declarative Python modules using dspy.signature, it efficiently implements PropertyToStateMapper to parse specifications and output corresponding mappings as a dict.

[1] Note: These are sanitized illustrations to protect proprietary information.

```python
class SpapiTester:
    def __init__(self):
        self.prop_to_state = dspy.TypedChainOfThought(
    PropertyToStateMapper)
        self.gen_test_obj = dspy.TypedChainOfThought(
    TestObjGenerator)
        self.method_tester = MethodTester()
    def forward(self,api_obj,mapping_spec,method_type):
        # LLM as parser
        mapping = self.prop_to_state(api_obj,
    mapping_spec)
        # LLM as fuzzer
        test_obj = self.gen_test_obj(mapping)
        # Conventional automation
        test_case = self.method_tester(test_obj)
        return test_case
```

Listing 1: spapi-tester: polymer for spapi testing.

Upon translation, the task of writing a test case remains. The conventional option is to use a REST API fuzzer [9] but, to avoid the cost of selecting and adapting one, spapi-tester uses the skeleton-key LLM itself as the fuzzer. It defines TestObjGenerator to (a) take the LLM-assembled mapping between endpoint properties like acMode and corresponding vehicle states like ACState, and (b) asks the LLM to choose a corresponding pair like acMode:STANDARD and ACState:1 as test objects. Finally, there remains the step of using the generated objects to write the test case and test the endpoints and these steps are quite simple to conventionally automate.

Applying polymer, spapi-tester polymerized one information flow (property to state mapping) and one product task (writing test cases) within a short span of 2 months, automating 2-3 FTEs worth of effort. Following a successful pilot to confirm its quality and utility [24], Volvo is currently deploying spapi-tester. Though this brings valuable gains, the coordination burden remains largely intact.

4 IMPLEMENTATION WORKFLOW AS CODE

Encouraged by the relative ease with which polymer automates testing, we consider left shifting further by applying it to the implementation workflow. Naturally, the complexity of translating a specification into a working implementation is of a higher order. Take spapi, where implementing an endpoint like /climate with a property like acMode, requires synthesis of methods get_acMode and put_acMode. The workflow involves three key steps — discovering the corresponding vehicle state ACState, discovering corresponding CAN signals APIACMode* to read/write state, and using this information to implement get and put methods. Discovering correspondences in a state of under-specification, as we previously noted, requires intense coordination. Volvo estimates that completing these three steps for hundreds of spapi APIs requires 15-20 FTEs. Applying polymer, and leveraging the skeleton key LLM, spapi-coder[1] (Listing 2) bypasses manual coordination and automates endpoint implementation by treating it as a case of program synthesis.

Using skeleton-key LLMs, spapi-coder addresses all three dimensions [11] of program synthesis — (a) user intent, (b) a search space, and (c) a search technique. To synthesize REST APIs like /climate, spapi-coder uses OpenAPI specifications to capture user intent. The synthesize_property function takes the OpenAPI property definition as input to synthesize its GET and/or PUT methods. Since endpoint methods access state by reading and writing CAN signals,

```
class SpapiCoder:
    def __init__(self):
        self.signal_synthesizer = SignalSynthesizer()
        self.signal_retriever = SignalRetriever()
        self.prop_synthesizer = dspy.TypedPredictor(
    PropertySynthesizer)
        self.template_examples = fetch_examples()
    # LLM + conventional program synthesis
    def synthesize_signals(self, state_def):
        rd_signal, wr_signal = self.signal_synthesizer(
    state_def)
        self.signal_retriever.insert_signal(rd_signal)
        self.signal_retriever.insert_signal(wr_signal)
    # RAG-based program synthesis
    def synthesize_property(self, property, k=5):
        top_k_signals = self.signal_retriever.k_relevant(
    property["description"], k)
        return self.prop_synthesizer(top_k_signals, self.
    template_examples)
```

Listing 2: `spapi-coder`: `polymer` for `spapi` implementation.

the search space includes all possible CAN signals corresponding to vehicle states. In control applications, it is often possible to extract the definition of a state like `ACState` from the code. The `synthesize_signal` function takes the definition of a state like `ACState` to synthesize read/write `APIACMode*` signals. To search over this space and synthesize endpoint methods, `spapi`-coder uses Retrieval-augmented Generation (RAG). By loading all signal definitions into a vector database, and using the property definition as the query, `synthesize_property` searches for the k most relevant signals and feeds it into an LLM call defined by `PropertySynthesizer`. Using the most relevant signals, and some few-shot examples, this LLM call synthesizes get and put methods for accessing state. This bridges all disciplinary gaps, fixes under-specification, and polymerizes endpoint synthesis, eliminating the coordination vortex.

Leveraging `polymer` and the skeleton-key LLM, `spapi`-coder polymerizes a workflow consisting of three tasks and the vortex of information flows that interconnect them. Unlike `spapi`-tester, which is currently being deployed, `spapi`-coder is currently in a pilot phase. Volvo is seeing encouraging signs that it helps automate `spapi` implementation, potentially streamlining 15-20 FTEs worth of effort.

5 OPEN CHALLENGES

Despite its adoption at Volvo, saving 2-3 FTEs in one case and promising to save 15-20 FTEs in another, challenges to unlocking `polymer`'s potential remain. Here are a few.

What to `polymer`-ize?: `spapi` turns out to be a successful showcase of `polymer` because (a) there is a clear case of inflated complexity, and (b) there is a pre-existing decomposition which can be SW-defined and polymerized using skeleton-key LLMs. Such clarity may be absent in other real-world workflows, so organizations need to carefully assess whether `polymer` is suitable for a given workflow and which of its tasks and transitions should involve LLMs. How does one efficiently and effectively make this decision? What risk versus gain calculus does one use to ease this decision-making?

Back To the Future: The adoption of `polymer` reintroduces an old problem: companies must validate the `polymer` processes they implement. How can one ensure that `polymer` processes reliably produce SW artifacts at scale, and build sufficient trust among stakeholders? What do test cases for `polymer` processes even look like? How does the test oracle problem change if the system under test is a development workflow? How do we develop workflow automation oracles, and how much human involvement do they require?

Agile Change Management: On the one hand, `polymer` will revolutionize change management. Since `polymer` processes are executable and can be simulated, they allow management to experiment with org structures at low cost. It will help them answer questions like — How do we ensure that the organization is resilient enough to withstand market shocks while also being responsive to new opportunities? On the other hand, adopting `polymer` will itself require substantial change management. The technology is still new to many stakeholders, and it is still maturing. How can organizations progressively transition to `polymer` while adequately addressing these limitations, and also continuing to deliver customer value?

Bridging the Engineer-MBA Gap: Today, in cases like `spapi`, engineers focus on development while managers concentrate on market demands, cost efficiency, and strategic planning. Given the intertwined nature of management and development, can `polymer` help break this strict separation to better integrate management and development activities in a workflow? In such a converged state, can it ensure that decision-making seamlessly incorporates both technical and business insights to identify and meet market needs?

We call upon the SE community to engage in research and collaborative efforts to address these open questions.

6 RELATED WORK

Development workflows as software offer a novel approach to Software Process Improvement (SPI) [16]. Unlike traditional process-focused SPI methods, `polymer` treats processes as software systems, making monitoring, feedback, iterative refinement easier at scale. `polymer` is a new form of Model-Driven Engineering (MDE) [7] that leverages LLMs, as skeleton keys, to seamlessly formalize and model existing stakeholder processes rather than requiring the disruptive adoption of a new process and the domain-specific language that usually accompanies it. Further, as shown in our example and in recent studies [10, 17, 23], LLMs are well-suited for producing software. `Polymer` offers a systematic framework for integrating LLMs with conventional automation to achieve this. An important framework that is parallel to our proposal is multi-agent systems [12], which orchestrates collaborative LLM agents to solve SE tasks. While autonomous solution discovery and evaluation are powerful, `polymer` provides a complementary path for cases where task decomposition and solution spaces are better known. Not only can this speed up task solving, it can also ease the evaluation of solutions. Additionally, the deliberate design of development workflows, merging LLM use with conventional automation, applies the core principles compound AI systems [25] to automate SW development.

7 CONCLUSION

We have presented `polymer`: a promising new approach for automating development workflows. By decomposing workflows and combining LLMs with conventional automation, `polymer` writes development tasks and transitions as code. Applying `polymer` at Volvo achieved practical cost savings. A wider application to fully realize its potential raises open questions that require further research. We invite the community to join us in answering them.

REFERENCES

[1] Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1946–1969. https://doi.org/10.1145/3591300

[2] Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=pXaEYzrFae

[3] Betsy Beyer, Niall Murphy, David Rensin, Stephen Thorne, and Kent Kawahara. 2018. *The Site Reliability Workbook*. https://landing.google.com/sre/book.html

[4] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2024. Evaluating LLMs for Hardware Design and Test. *CoRR* abs/2405.02326 (2024). https://doi.org/10.48550/ARXIV.2405.02326 arXiv:2405.02326

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[6] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL] https://arxiv.org/abs/2303.12712

[7] Alberto Rodrigues Da Silva. 2015. Model-driven engineering: A survey supported by the unified conceptual model. *Computer languages, systems & structures* 43 (2015), 139–155.

[8] Alfonso Fuggetta and Elisabetta Di Nitto. 2014. Software process. In *Future of Software Engineering Proceedings*. 1–12.

[9] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–41.

[10] Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X Olausson, Muxin Liu, Joshua B Tenenbaum, and Jacob Andreas. 2023. Lilo: Learning interpretable libraries by compressing and documenting code. *arXiv preprint arXiv:2310.19791* (2023).

[11] Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

[12] Junda He, Christoph Treude, and David Lo. 2024. LLM-Based Multi-Agent Systems for Software Engineering: Vision and the Road Ahead. *arXiv preprint arXiv:2404.04834* (2024).

[13] Gonzalo Jaimovitch-López, Cèsar Ferri, José Hernández-Orallo, Fernando Martínez-Plumed, and María José Ramírez-Quintana. 2023. Can language models automate data wrangling? *Machine Learning* 112, 6 (2023), 2053–2082.

[14] Aishwarya Kamath and Rajarshi Das. 2018. A survey on semantic parsing. *arXiv preprint arXiv:1812.00978* (2018).

[15] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *CoRR* abs/2310.03714 (2023). https://doi.org/10.48550/ARXIV.2310.03714 arXiv:2310.03714

[16] Marco Kuhrmann, Philipp Diebold, and Jürgen Münch. 2016. Software process improvement: a systematic mapping study on the state of the art. *PeerJ Computer Science* 2 (2016), e62.

[17] Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. 2023. Fully autonomous programming with large language models. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1146–1155.

[18] Liane Makatura, Michael Foshey, Bohan Wang, Felix Hähnlein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Elaine Owens, Peter Yichen Chen, Allan Zhao, Amy Zhu, Wil J. Norton, Edward Gu, Joshua Jacob, Yifei Li, Adriana Schulz, and Wojciech Matusik. 2024. Large Language Models for Design and Manufacturing. *An MIT Exploration of Generative AI* (mar 27 2024). https://mit-genai.pubpub.org/pub/nmypmnhs.

[19] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).

[20] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-Aligned Decoding. arXiv:2405.21047 [cs.AI] https://arxiv.org/abs/2405.21047

[21] David Schlangen. 2024. LLMs as Function Approximators: Terminology, Taxonomy, and Questions for Evaluation. arXiv:2407.13744 [cs.CL] https://arxiv.org/abs/2407.13744

[22] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. SynCode: LLM Generation with Grammar Augmentation. arXiv:2403.01632 [cs.LG] https://arxiv.org/abs/2403.01632

[23] David Vella Zarb, Geoff Parks, and Timoleon Kipouros. 2024. Synergistic Utilization of LLMs for Program Synthesis. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 539–542.

[24] Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. 2025. Automating a Complete Software Test Process Using LLMs: An Automotive Case Study. *arXiv preprint arXiv:2502.04008* (2025).

[25] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.