Automated Service Design with Cerulean Project Showcase

Vaastav Anand
MPI-SWS
Germany

Alok Gautam Kumbhare
Microsoft
USA

Celine Irvene
Microsoft
USA

Chetan Bansal
Microsoft
USA

Gagan Somashekar
Microsoft
USA

Jonathan Mace
Microsoft
USA

Pedro Las-Casas
Microsoft
USA

Rodrigo Fonseca
Microsoft
USA

Motivation

Modern cloud applications are commonly developed as microservice systems [2]. In these systems, the application is decomposed into loosely-coupled lightweight services connected over the network, each responsible for providing a single high-level business capability in the system.

Automating the design, operation, and optimization of these systems, has been a longstanding [3], but also an elusive goal. This is due to two key contributing factors. First, there was a lack of standardization of tools that can help develop, implement, and improve systems. Second, there was no way to automatically convert high-level requirements of the system into corresponding functional implementations, leading to long and arduous engineering processes that often leave gaps.

However, recent trends makes it feasible to revisit these goals: the rise of standardization tools, and of Large Language Models (LLMs). The rise of standardization tools like Kubernetes [7] for deployment, Blueprint [1] and ServiceWeaver [4] for development and implementation generation, and OpenTelemetry [6] for observability, have provided a backbone that developers can use to build robust systems. LLMs, on the other hand, provide a way of representing user requirements at a high-level, in natural language, as well as generative abilities that can be used to develop applications from the high-level descriptions.

In this project showcase, we introduce Cerulean, a modular, extensible, human-in-the-loop system that combines standardized tools with the expressive power of LLMs to automatically generate implementations of microservice systems.

Challenges

While standardized tools and LLMs provide the basic blocks for enabling automating system design and implementation, there still remain a variety of challenges to use these techniques in conjunction, due to the nondeterministic nature LLMs. We highlight these fundamental challenges below.

Lack of Correctness guarantees. The knowledge base of LLMs may consist of buggy, incorrect, and poorly written code that can degrade the quality of the generated code. Correctness of the generated code is hard to check and it grows in difficulty with the increase in the size of the generated code. Moreover, if we generate both implementations and tests using LLMs, then we might encounter situations where both tests and code might be incorrect.

User requirements mismatch. LLMs are well-known to suffer from faithful hallucinations, especially instruction inconsistency [5], in which LLMs deviate from user directives. This can result in generated code not containing the features desired by the users. This can also potentially lead to an unexpected design being used by the LLMs for the desired system.

Lack of explainability. Developers often find it hard to understand large foreign codebases. Trying to understand large amounts of code generated by LLMs poses the same problem as developers now need to understand code produced by an AI model. Failing to correctly understand the generated code and blindly copy-pasting the code can lead to bugs and monetary loss.

Cerulean Design

Cerulean combines LLM-based generation techniques with existing techniques from software development cycle to ensure that the generated system is correct. Cerulean manifests this combination through the process of *hierarchical generation*, comprising generation at multiple abstraction levels. Each level generates a validated representation of the user requirements that is used as input for generating artifacts in the next level of abstraction.

Figure 1 provides an overview of our current working prototype of hierarchical generation, implemented in $3k$ lines of python code. The prototype currently generates microservice systems in Go. Cerulean starts the generation procedure at the highest level of abstraction of a system and then iteratively drills down deeper into increasingly lower levels of abstraction. At each level, Cerulean provides a validation procedure that validates the LLM-generated output as well as explainability artifacts that can help users reason about the generated system. The hierarchical generation process is modular and can be extended with other components that provide stronger guarantees such as formal verification. Below, we provide details about the generation procedure of our current prototype.

Input Requirements. Cerulean expects the user to provide as input the functional and behavioral requirements of the desired system. These include a brief overview description of the system and user stories describing the functional behavior of the system.

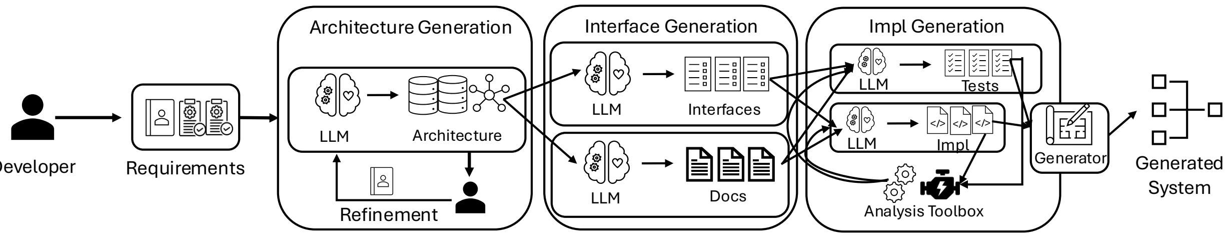

Figure 1: **Hierarchical generation for automated system design**

Architecture Design. Based on the provided requirements, Cerulean prompts the LLM to generate the high-level architecture of the system. To provide explainability, Cerulean converts the generated list of modular components into a dependency graph to show to the operator. The operator validates the generated design or optionally provides a critique to refine the architecture design, which is used by Cerulean to generate an updated design. This operator-in-the-loop critique-based refinement of the generated design continues until the operator is satisfied with the generated design.

Next, Cerulean augments the dependency graph with databases and caches, generates schemas for the databases, and connects existing components to the newly added databases and caches. To ensure correctness of the generated schemas, Cerulean uses operator-in-the-loop critique-based refinement.

Interface Design. At the next level of abstraction, Cerulean generates the interfaces of the components defined in the dependency graph from the previous phase. The generation proceeds in reverse topological order of the dependency graph. The generated interface for each component contains a list of functions and their corresponding function signatures that can be used by other components for interaction. For each component, Cerulean includes the previously generated interfaces of the dependencies of the component to ensure that the generated interface correctly provide pathways into the dependent components.

To ensure correctness, the operator can optionally do critique-based refinement to add or remove functions in each interface. In this phase, Cerulean provides explainability in two ways. First, for each interface, documentation is generated that describes what the interface does at a high level, descriptions for individual functions in the interface, and an implementation plan consisting of the list of all outgoing calls to dependencies for each function. Second, the generated interfaces are combined with the previously existing dependency graph to generate a UML Class diagram to combine the generated interfaces with the dependency graph.

Implementation Generation. The implementation generation process follows the principle of test-driven development. To instantiate test-driven development, the generation process is broken down into three individual phases. In the first phase, Cerulean uses the documentation, interfaces, and the implementation plans generated in the previous level to generate unit tests for each function in each component's interface. The generated unit test uses mock objects to handle dependencies of the component. In the second phase, the process once again uses the previously generated documentation and interface to generate the implementation of each function. To ensure that the generated code is valid, Cerulean executes static analysis passes on the generated code. If the generated code passes the static checks then Cerulean proceeds to the next phase. If the static checks fail, then Cerulean uses the error output as part of the refinement context to generate a new implementation with the bugs fixed. This continues until the static checks on the code pass. In the final phase, Cerulean refines the generated tests and implementations. To do the refinement, Cerulean first runs the generated tests against the generated implementation with code coverage reporting enabled. If all tests do not pass, then Cerulean gathers the test failure output and provides that as context along with the generated implementation and test to the LLM to improve the implementation and/or test such that all tests pass. This refinement procedure continues until either all tests pass or Cerulean exceeds a maximum number of retries.

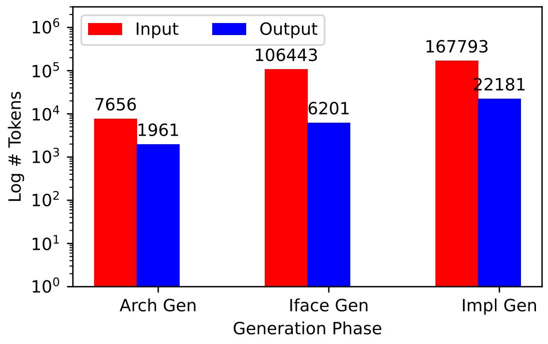

Figure 2: **Cost of using Cerulean**

Initialization Generation. In the last phase, Cerulean generates the scripts needed to initialize the system. These include scripts to correctly setup the databases, scripts to setup the initial state of the various components, and the scripts to apply the generated behavioural properties (observability, security, etc) on to the generated system. To apply the behavioral properties, we convert the generated implementations and behavioral properties into the input specifications of the Blueprint [1] compiler as their abstractions allow us to transparently add features to a microservice application.

Preliminary Results. To test out the efficacy of Cerulean, we prompt Cerulean to design a shoe-sharing application where users can loan their shoes to other people. We choose this application as it is a unseen application from the perspective of the LLMs and unlikely to be in its training data. The initial requirements are provided in 164 words.

To generate the system, Cerulean took 12.7 minutes and generated 6 user-defined services backed by 5 different databases. In total, the system contained 30 different endpoints across the 6 user-defined services. For the 30 different endpoints, Cerulean generated 43 different unit tests with a total test success rate of 76.7% when executed against the generated implementation. The generated tests achieved a 100% test coverage for 18 of the 30 endpoints, with total coverage of 90.2%. In total, Cerulean made 140 calls to the LLM to generate the system. Figure 2 shows the total number of tokens input by Cerulean to the various LLM calls and the total number of output tokens generated.

References

[1] V. Anand, D. Garg, A. Kaufmann, and J. Mace. Blueprint: A toolchain for highly-reconfigurable microservice applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 482–497, 2023.

[2] A. Cockcroft. Microservices workshop: Why, what, and how to get there. (April 2016). Retrieved October 2020 from https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference, 2016.

[3] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM systems Journal*, 42(1):5–18, 2003.

[4] S. Ghemawat, R. Grandl, S. Petrovic, M. Whittaker, P. Patel, I. Posva, and A. Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 110–117, 2023.

[5] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232*, 2023.

[6] OpenTelemetry. High-quality, ubiquitous, and portable telemetry to enable effective observability. Accessed August 2024 from https://opentelemetry.io/.

[7] E. Shanks. Kubernetes - desired state and control loops. Accessed July, 2024 from https://theithollow.com/2019/09/16/kubernetes-desired-state-and-control-loops/, 2019.