GARMA: Generative Architectural Resource Demand Estimation for Microservice Applications

Maximilian Hummel
ABB AG Corporate Research Center, Germany
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
maximilian.hummel@kit.edu

Dominik Fuchß, Sophie Corallo, Nathan Hagel, Minakshi Kaushik, Jan Keim, Ralf Reussner
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{name.lastname}@kit.edu

Heiko Koziolek
ABB AG Corporate Research Center, Germany
heiko.koziolek@de.abb.com

***Abstract*—Architectural performance models, such as the Palladio Component Model, can support early design decisions for microservice systems by enabling performance simulation. However, early stage models often lack the service resource demand specifications (e.g., 100 ms CPU demand) required for such performance simulations. Existing approaches to build performance models often depend on late-stage running prototypes and intrusive profiling, or otherwise, on manual expert estimation, which is costly and hard to reproduce. We present GARMA, an LLM-based workflow that processes early design artifacts and automatically generates behavioral microservice models with bounded best-case and worst-case resource demand estimates. In a microservice test scenario (TeaStore), GARMA generated 150 behavioral models that closely matched reference structures (average Jaccard similarity 0.97; perfect matches at 84%). The predicted CPU resource demand intervals aligned well with measurements, capturing most (85%) user-facing interactions within the predicted intervals. In addition, GARMA produced consistently narrow CPU-demand intervals for the main user-facing steps. Additionally, we compared GARMA against a naive LLM baseline that estimates CPU demands without performing the presented LLM workflow. GARMA consistently outperformed that baseline, achieving lower absolute errors, higher coverage, and narrower intervals.**

***Index Terms*—Software architecture, microservices, cloud-native systems, architectural performance modeling, Palladio Component Model, resource demand estimation, LLMs**

I. Introduction

In cloud-native systems, microservices play a crucial role. Responsiveness (as perceived by users) as well as resource efficiency (on the server side) are decisive factors for commercial success. The reason is that the time behavior of a cloud-native system depends on the complex interactions between the services, runtimes, and workloads of its microservices [1]. As these interactions impact design choices early on, architects benefit from comparing alternatives before implementation.

Fortunately, design alternatives can be explored prior to implementation and deployment using architectural performance models, such as the Palladio Component Model (PCM) [2]. However, PCM requires, for each operation in a microservice, the specification of resource demands (e.g., CPU demands), and these values may be difficult to obtain during early design [3], [4]. The reason is that resource demands depend on many interacting factors such as application logic, workload, and runtime. During early design, such factors are often not known in sufficient detail. Furthermore, a running system is often not even available for measurements. Architects are thus constrained to make structural decisions while lacking quantitative guidance for reasoning about performance trade-offs (e.g., in component decomposition).

Existing measurement-based techniques for resource demand estimation [5]–[8] require a running microservice application, while profiling approaches require intrusive instrumentation. Therefore, measurement-based techniques are not applicable when no executable system is available in the early design. Estimation-based techniques require a performance engineer to manually make informed best-/ worst-case predictions of a microservice's resource demands based on available specifications and knowledge of the anticipated deployment environment [3]. This manual process is time-consuming, difficult to reproduce, and hard to scale to complex, heterogeneous microservice applications [9]–[11]. Informal design sketches or textual descriptions useful for resource demand estimation may be available in early stages, but were inaccessible to algorithmic processing before the advent of LLMs.

This paper introduces GARMA, an LLM-based workflow that processes artifacts available during early design stages and automatically generates PCM Resource Demanding Service Effect Specifications (RDSEFFs) with estimated resource demands for microservice operations. GARMA requires a so-called workload dependency graph sketched during early design and textual specifications of workload characteristics and the runtime environment as inputs, and uses an LLM to estimate the best- and worst-case resource demands. In the established performance modeling process by Connie Smith [12], execution graphs are derived from critical use case scenarios before quantitative analysis, making such a workload dependency graph already available at this stage. Nevertheless, conducting this quantitative performance analysis requires architects to manually estimate or measure resource demands, which requires considerable expertise. GARMA solves this problem by deriving resource demand estimates from the artifacts produced prior to the quantitative performance evaluation step in the performance modeling process. GARMA consists of three steps: (i) normalize heterogeneous input

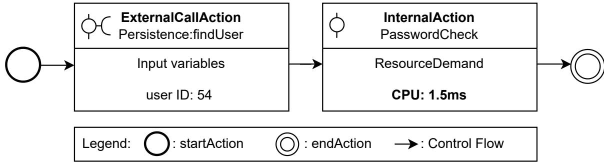

Fig. 1. RDSEFF for TeaStore `auth:login`: External call to `persistence:findUser` & internal action `PasswordCheck` with CPU demand.

into performance-relevant information, (ii) derive structural RDSEFF models, and (iii) estimate resource demands using a bounded-uncertainty strategy following Smith et al. [3].

In TeaStore (five microservices), GARMA yields higher accuracy and tighter intervals than a naive LLM baseline.

II. Background

The *Palladio Component Model (PCM)* provides software architects with a tool to predict quality attributes such as response time, throughput, and resource utilization before implementation [2]. PCM separates concerns into components, allocation, resource environment, and usage models, enabling systematic experimentation with design alternatives.

In the PCM, a *Resource Demanding Service Effect Specification (RDSEFF)* represents a single operation of a component's provided interface (see Figure 1). In this work, components correspond to microservices and operations to API endpoints. An RDSEFF models the endpoint's internal control flow as an abstracted graph comprising *InternalActions* (computations with resource demands), *ExternalCallActions* (calls to other services), *BranchActions*, and *LoopActions*. Resource demands are expressed via Performance Model Parameters (PMPs). In the following, we denote the CPU resource demand of an operation i as S_i, defined as the workload-independent CPU time per invocation of that operation. Thus, S_i captures the pure processing time required to serve a single request, excluding effects of arrival rate, concurrency, or queuing.

A PMP attaches a typed, unit-carrying expression representing per-invocation resource demand to an action (e.g., CPU: 1.5ms in Figure 1), optionally parameterized by variables (e.g., request size). PMPs can represent resource demands (e.g., CPU) for InternalActions, loop iterations for LoopActions, or input variables for ExternalCallActions.

An example of a RDSEFF with PMPs is shown in Figure 1.

Estimating resource demands in early design stages is difficult [3]. Software architects often lack precise information on CPU time, storage access, or network activity each microservice will require during execution. To create performance models during early design, architects need at least approximate resource demand values.

Such values are difficult to characterize with high accuracy. Requirements may change, microservice behavior is only abstractly defined, and no running system allows measurements. Given this uncertainty, manually specifying best-case and worst-case estimates of resource demands is recommended [3]. Such bounded estimates help architects anticipate potential bottlenecks; for example, in Figure 1, a best-case CPU demand for `PasswordCheck` may assume a short hash comparison on warm caches, whereas a worst-case estimate accounts for cold-cache behavior, indicating whether this `PasswordCheck` could dominate the response time.

In summary, architects need support in translating early-stage design descriptions, such as microservice structure, initial API code, and textual specifications into behavioral RDSEFF models with PMPs estimates that explicitly account for uncertainty. This assistance is provided by GARMA, generating structured RDSEFFs enriched with PMPs.

III. Related Work

In practice, software engineers have manually estimated resource demands for software systems informally for many decades [13] to cope with limited computing resources, or to improve system performance systematically. Connie Smith established the discipline of software performance engineering (SPE) in the 1980s [14]. She demonstrated how software engineers can build execution graphs representing software processing steps in early performance walkthroughs and "guess" the best- and worst-case resource demands and invocation frequencies based on assumed functionality and experience [12]. This requires deep expertise, is time-consuming, and potentially error-prone. However, this approach enables performance predictions before the system is implemented, thereby informing decisions on early, fundamental architectural design choices that are difficult to change later.

Measurement-based resource demand estimation approaches assume that a software system is already built or partially built, or that at least prototypes of performance-sensitive functionality can be executed and measured to derive resource demands. For example, the Kieker approach [8] provides an application performance monitoring (APM) framework for capturing the information required to estimate resource demands. Commercial APM solutions, such as Dynatrace, AppDynamics, and Datadog can support this work as well. "ByCounter" from Krogmann et al. [7] abstracts from concrete timings and instead counts bytecode instructions as platform-independent resource demands.

Statistical estimation techniques for resource demands have been collected by Spinner et al. [5]. Among them are ordinary least squares regression, service demand law, maximum likelihood estimator, and Kalman filters. Zheng, Woodside, and Litoui use the latter for online demand estimation of performance models [15]. Grohmann et al. [6] propose a self-tuning approach for resource-demand estimation techniques. None of these approaches offers support for estimating resource demands without executable software.

In addition to estimating individual resource demands, there are approaches to **construct entire performance models**, including components and deployment models [16]. Kappler et al. [17] propose Java2PCM to derive PCM RDSEFFs statically from Java source code. Krogmann et al. [7] employ a genetic search to derive parameter dependencies in PCM

RDSEFFs from source code. Garbi et al. [18] use recurrent neural networks to mimic service and arrival rates of queuing networks. As Hummel et al. [19] also show in their recent work, none of these studies used LLMs for the task.

IV. GARMA

At a high level, GARMA consumes (i) a software description, (ii) workload characteristics, and (iii) a runtime environment specification, processes them through a sequence of LLM invocations, and produces RDSEFFs enriched with PMPs.

GARMA constructs RDSEFFs and PMPs based on these early design artifacts for PCM simulation without requiring a deployed system. Figure 2 provides a high-level overview of the LLM-based workflow, which comprises four phases: (1) input normalization, (2) architectural behavior modeling, (3) performance parameterization, and (4) validation.

A. Inputs

GARMA classifies its inputs into three categories, aligned with the information required for resource demand estimation as defined by Connie U. Smith et al. [12].

(i) A **software description** outlines the operations responsible for handling system events. (ii)**Workload characteristics** identify the different types of events that occur in the system and quantify their intensity. (iii)The **runtime environment** comprises the hardware components and key software services required for system execution, along with their service rates.

As a software description, GARMA expects a workload dependency graph, a directed graph representing the microservice architecture and the request propagation through microservices. Nodes represent operations (API endpoints, e.g., `auth:login`); edges represent invocation dependencies. Such a graph is typically present during early stages in the established performance engineering processes to define critical use cases and request paths. Therefore, this workload dependency graph represents a performance relevant slice of the whole application and can be considered available at this stage. For all operations, GARMA derives RDSEFFs (see Figure 1) and CPU demands as PMPs. API-level code can be provided for each operation. In our setting, API-level code refers to the high-level endpoint handlers that implement the control flow of an operation, enriched with inline comments that expose performance-relevant internals. For example, the `auth:login` handler performs request-parameter extraction, an external call to `persistence:findUser`, and CPU-intensive cryptographic steps such as password verification, followed by object manipulation and JSON serialization. These annotated code snippets thus contain both the structural invocation sequence in the workload dependency graph and the algorithmic information needed for deriving RDSEFF actions and their associated PMPs. GARMA accepts all programming languages as it uses an LLM for code processing.

Although GARMA can leverage API-level code for individual operations, these snippets still reflect early design and do not constitute a runnable or profiled system. The code lacks deployment and integration context and is not embedded in a configured workload and monitoring setup. Turning such code snippets into a measurable prototype would require additional engineering effort (e.g., containerization, configuration, test data, and instrumentation). GARMA instead treats code as a rich early-stage design artifact and uses it statically to derive RDSEFF structures and resource-demand estimates.

The workload characteristics and runtime environment must be described textually to GARMA. For example, the workload characteristic could contain a statement like "20 requests per second for the `auth:login` operation ..." The runtime environment could contain: "The TeaStore is deployed on Google Kubernetes Engine ..."

TABLE I
EXCERPT FROM A PERFORMANCE QUESTIONNAIRE FOR A LOGIN REQUEST

C1. Workload Dependency Graph
Q1. *What are the specific operations executed by each node in the workload dependency graph?*
A1. The login workload issues a `POST /login` request to `webui:login`, the `webui` ...
C2. Workload Characteristics
Q2. *What is the typical frequency for requests?*
A2. The workload issues 20 requests per second in total, and ...
... Remaining categories and questionnaire items are omitted for brevity.

B. Step 1: Input Normalization

The input normalization step uses a structured questionnaire to extract performance-relevant facts from the inputs.

We designed the performance questionnaire with 22 questions, using the performance-relevant categories for software performance engineering described by Smith et al. [3].

The questionnaire is auto-filled by an LLM from the available artifacts whenever explicit evidence is present. For each question, an LLM is invoked to answer the question based solely on the available input artifacts (see Section IV-A). If some information is missing, the LLM leaves the question unanswered. This step, therefore, transforms heterogeneous and informal input artifacts into a structured performance questionnaire. For example, for the TeaStore `login` request, the LLM's answer could look like in Table I.

C. Step 2: Architectural Behavior Modeling

The architectural behavior modeling step derives structural RDSEFF models from the workload dependency graph and the corresponding per-node code and can, in principle, process any programming language or even pseudocode due to the language-agnostic nature of LLM-based code understanding.

It follows an aggregation policy, that is embedded into the LLM prompt for this step: Local computations in one microservice (e.g., PasswordCheck in the `auth` microservice Figure 1) are aggregated into *InternalActions*. *BranchActions* and *LoopActions* that contain no external calls are combined

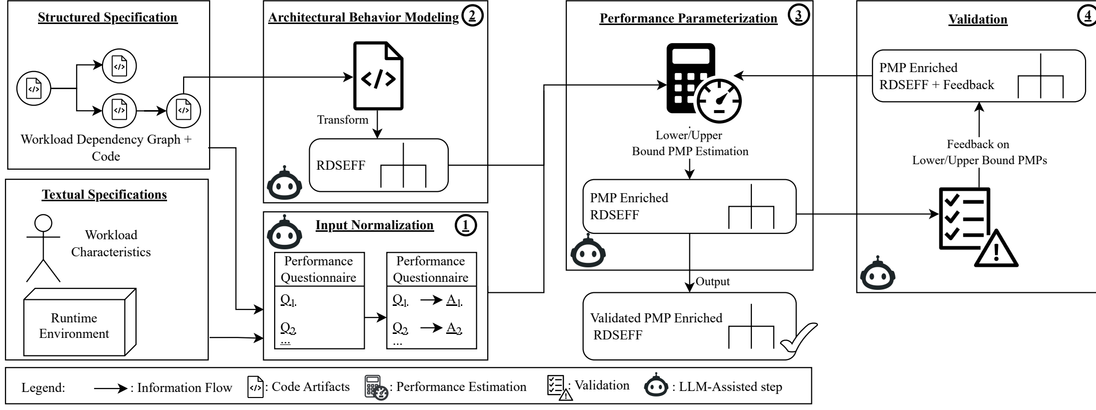

Fig. 2. GARMA workflow from input artifacts to Resource Demanding Service Effect Specifications (RDSEFFs) with Performance Model Parameters (PMPs).

into a single InternalAction. Thereby, GARMA avoids over-fragmentation. Calls to other microservices are represented as *ExternalCallActions*, e.g., the `auth:login` endpoint calling the `persistence:findUser` operation as shown in Figure 1. These external calls must be consistent with the workload dependency graph edges. Control-flow constructs (branches, loops) are modeled only when they contain external interactions. Applying this step to the `auth:login` node produces the RDSEFF shown in Figure 1.

D. Step 3: Performance Parameterization

The performance parameterization attaches PMPs to actions within RDSEFFs and enables simulation and sensitivity analysis (e.g., with PCM). In Figure 1, the InternalAction receives PMPs as CPU resource-demand annotation. An exemplary parametric resource demand for the running example is:

$$CPU_{demand}(\texttt{auth:login}) = 1.0 + 0.05 \times payloadSize(\text{ms}),$$

where *payloadSize* is a workload driver, such as the size of the login request. In this example, the PMP models how the CPU time of the `auth:login` endpoint increases linearly with the payload size. PMPs can be constants or depend on input parameters. The prompt instructs the LLM to use parametric expressions whenever a PMP within an RDSEFF scales with an input parameter, and constant values otherwise.

This design aligns with the stochastic expression language supported by the PCM simulation framework [2]. ExternalCallActions carry argument and data-size characterizations as well as parametric dependencies. LoopActions carry iteration counts as constants or parametric expressions. BranchActions carry probability distributions. The estimation uses a decomposition-and-aggregation routine: Operations are broken down, per-operation demands are estimated. Scale effects are encoded via parametric dependencies tied to workload drivers (e.g., number of elements, payload size).

To obtain best-case and worst-case values, GARMA requires an LLM call to decompose the API-level code of each action into elementary operations and to estimate best-case and worst-case per-operation CPU demand, following Smith's bounded-uncertainty strategy [3]. Best-case values assume favorable effects (e.g., cache hits, minimal contention), whereas worst-case values assume plausible adverse conditions (e.g., cache misses, higher contention). The LLM aggregates these per-operation estimates into total best- and worst-case CPU demand intervals for the respective action. For PMP estimation, we employ role-based prompting ("You are an experienced performance engineer…"). Given the structural RDSEFF model, the API-level code, and the information about workload characteristics and the runtime environment, the LLM derives PMPs for each action. For this estimation, we assume that representative resource-demand characteristics are sufficiently present in the LLM corpus. The outcome is a set of RDSEFFs enriched with PMPs that capture both the behavioral structure and the interval-based parameter estimates. This step triggers one LLM invocation per action in the RDSEFF. This granularity is intentional: resource-demand estimation is performed strictly at the action level rather than the node level in the workload dependency graph, ensuring that the estimation scope is narrow. Our replication package contains the full prompt templates and used LLM instructions [20].

E. Step 4: Validation

The validation step performs model-internal consistency checks on PMPs, e.g. worst-case values must be greater than or equal to best-case values or spreads must be meaningful given the workload and runtime environment. The validation step employs an LLM configured to act as an unbiased validator, analyzing each action's PMPs together with the answered questionnaire. For each RDSEFF action attached with PMPs, this validator produces feedback flagging issues, suggests corrective directions and prompts re-estimation.

F. Outputs: RDSEFFs and Performance Model Parameters

GARMA produces two complementary output artifacts: (i) RDSEFFs that capture the control flow for each node in

the workload dependency graph and (ii) PMPs that annotate these actions with quantitative resource demand estimations.

All quantitative PMPs are expressed as best-case and worst-case values, following the bounded-uncertainty semantics introduced in Section II. GARMA uses these intervals to represent plausible early-design behavior without requiring measurements. The resulting PMP-enriched RDSEFFs serve as input to a PCM simulation (see Section II).

V. Evaluation

For the evaluation of GARMA, we follow a structured Goal-Question-Metric (GQM) plan [21], [22].

Goal: Generate accurate and decision-useful architectural performance models from heterogeneous input artifacts, covering both the derivation of RDSEFF structures and the estimation of CPU resource demands.

Q1: *How correctly does GARMA fill out the questionnaire from the available input artifacts?*

For Q1, we measure the fraction of correctly prefilled questionnaire items during input normalization per operation (min/avg/max) and report the overall average prefill rate. Correctness is assessed via manual cross-checking against the provided input artifacts.

Q2: *How accurate is the generated RDSEFF structure compared to manually derived reference models?*

Each RDSEFF is transformed into an action sequence S. Structural similarity between generated and reference models is measured using Jaccard similarity [23] based on the longest common subsequence (LCS):

$$J = \frac{|\mathrm{LCS}(S_{\mathrm{gen}}, S_{\mathrm{ref}})|}{|S_{\mathrm{gen}}| + |S_{\mathrm{ref}}| - |\mathrm{LCS}(S_{\mathrm{gen}}, S_{\mathrm{ref}})|} \quad (1)$$

Actions are compared by type and, for *ExternalCallActions*, by operation identifier (e.g., `auth:login`). $J \in [0, 1]$, where 1 is a perfect match. We report the percentage of perfectly matching RDSEFFs ($J = 1.0$) per request type and overall, as in prior work for RDSEFF reconstruction [24].

Q3: *How accurate are the estimated CPU resource demands (PMPs) in terms of best- and worst-case predictions compared to empirical measurements?*

For Q3, we use two complementary metrics to evaluate the PMP estimation. First, we report the absolute error between the predicted and measured mean CPU times per operation. The predicted mean is the midpoint of the predicted interval (PI). Expressing it as *Pred.,Mean* $\pm \frac{1}{2}$, *width* shows both the central value and the sharpness of the predicted interval, where sharpness is defined as $\frac{1}{2}(worst - best)$ and thus reflects how narrow (informative) the interval is. This is essential, since coverage alone is trivial to achieve with sufficiently wide intervals. Second, we compute coverage as the proportion of measured CPU times y_i that fall within the predicted interval. The predicted interval $[best_i, worst_i]$ corresponds to the best-case and worst-case CPU-demand PMPs of the operation's *InternalActions*:

$$\text{Coverage (Cov.)} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[\mathrm{best}_i \leq y_i \leq \mathrm{worst}_i] \quad (2)$$

Because CPU measurements are taken at the node level of the workload dependency graph, the *InternalAction* PMPs of each RDSEFF are aggregated so that their total CPU demand matches the measured node-level demand. We also report whether the measured mean falls within the predicted interval (PI) for each operation (Mean $\in$ PI).

Q4: *How much modeling effort is required to obtain performance models using GARMA compared to traditional performance engineering processes?* For Q4, we qualitatively assess the reduction in manual effort using prior user studies creating RDSEFFs and defining PMPs.

We evaluate GARMA with the TeaStore microservice application [25]. We executed controlled load tests to collect CPU time measurements and manually created RDSEFFs for these scenarios to evaluate the architectural behavior modeling.

The investigated scenario covers four sequential user-facing HTTP requests ((startPage, login, cartAdd, logout) in the **TeaStore** microservice application [25] deployed on Google Kubernetes Engine (GKE). All TeaStore services are deployed as pods on GKE with CPU requests and limits of 0.8 CPU cores per pod and no explicit memory limits. The cluster uses `e2-standard-4` worker nodes (4 vCPUs, 16 GB RAM) in a european region with default GKE networking.

The workload follows a constant-arrival-rate pattern with 20 requests/s. Each request contributing 25% of the total request mix, without think times. For each operation we recorded the thread CPU times. We used them as ground truth for assessing the quality of the predicted best- and worst-case CPU demands.

We modeled 14 RDSEFFs for the different TeaStore operations to compare with the generated RDSEFFs of GARMA. The workload dependency graph was constructed based on the TeaStore JAX-RS API code, with enriched inline comments for each called method where the code is located in another script and therefore not present in the API code. These inline comments explain high-level execution steps of the called methods. Such annotated API-level artifacts are consistent with the artifacts available after deriving execution graphs from critical scenarios in the SPE process by Smith [12], and can therefore serve as input for early resource-demand estimation.

For comparison, we introduce a single-pass naive LLM estimator that directly maps code, workload, and runtime descriptions to CPU demand estimates without constructing RDSEFF structures or per-action estimation. It uses the same bounded-estimation strategy (best- and worst-case intervals).

A. Results

We used GPT-5 with temperature 0, and ran the workflow in auto mode without a human in the loop for information enrichment. For each operation, we generated ten models to analyze the nondeterministic behavior of LLM output. In summary, we created and calibrated 150 RDSEFFs.

The following shows insights into the analysis of the results.

a) Q1 (Prefilling effectiveness): GARMA auto-fills 76.1% of questionnaire items across all operations in the TeaStore. A manual audit confirmed that prefilled answers

TABLE II
TEASTORE PMP EVALUATION: GARMA VS. NAIVE BASELINE (MS)

			GARMA				Naive Baseline			
		Mean ± Std	Pred. Mean	Abs. Err.	Mean ∈ PI	Cov.	Pred. Mean	Abs. Err.	Mean ∈ PI	Cov.
StartPage	webui:StartPage	2.58 ± 1.14	2.44 ± 1.18	**0.14**	✓	**86.4%**	5.75 ± 2.25	3.17	✗	14.5%
	image:webImages	0.43 ± 0.13	0.71 ± 0.28	0.28	✓	**44.8%**	0.15 ± 0.08	0.27	✗	4.2%
	auth:isLoggedIn	0.18 ± 0.09	0.27 ± 0.12	**0.09**	✓	**49.6%**	0.46 ± 0.26	0.28	✗	27.1%
	persist:listCategories	0.50 ± 0.12	0.92 ± 0.46	0.42	✓	55.1%	0.40 ± 0.20	**0.10**	✓	**83.7%**
Login	webui:login	1.26 ± 0.53	0.88 ± 0.42	**0.38**	✓	**74.1%**	4.06 ± 1.69	2.80	✗	6.4%
	auth:login	7.77 ± 0.81	7.11 ± 3.70	**0.66**	✓	**99.8%**	6.15 ± 2.65	1.62	✓	87.4%
	persist:findUser	0.69 ± 0.14	0.51 ± 0.27	**0.18**	✓	**79.3%**	5.20 ± 2.70	4.51	✗	0.0%
CartAdd	webui:cartAdd	1.74 ± 0.80	1.77 ± 0.90	**0.03**	✓	**90.1%**	5.20 ± 2.70	3.46	✗	11.1%
	auth:cartAdd	1.54 ± 0.74	1.04 ± 0.38	**0.50**	✗	**66.6%**	3.09 ± 1.73	1.56	✓	37.0%
	persist:findProd	0.30 ± 0.16	4.07 ± 2.31	3.77	✗	0.0%	0.16 ± 0.11	**0.14**	✗	**61.2%**
Logout	webui:logout	1.80 ± 0.89	1.20 ± 0.67	**0.60**	✓	**75.7%**	0.77 ± 0.42	1.03	✗	7.7%
	auth:logout	0.20 ± 0.10	0.07 ± 0.04	**0.13**	✗	**15.0%**	2.39 ± 1.19	2.20	✗	0.0%

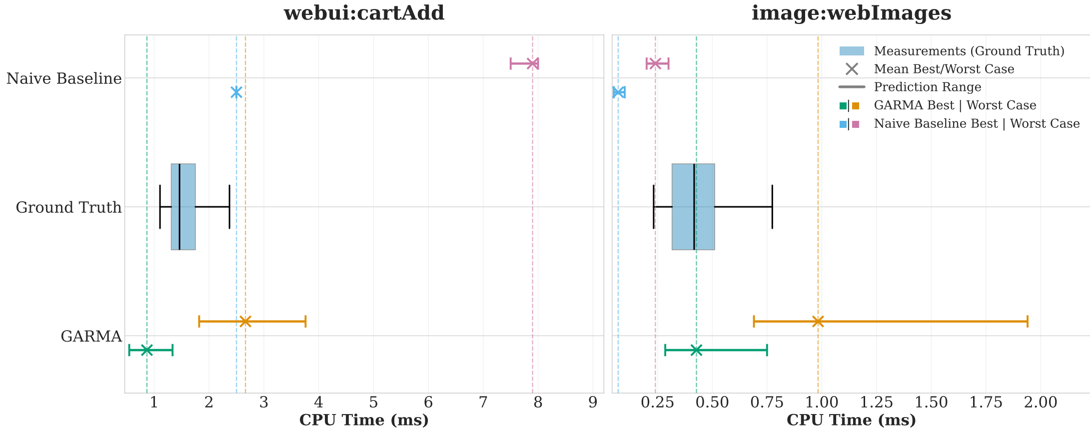

Fig. 3. TeaStore comparison between average predicted interval and measurements for specific endpoints in the image- and webui-service

matched available specifications, unanswered questions correctly reflected information gaps in the inputs. For each of the 14 operations, we checked the answers once and observed consistent answers across different runs.

b) Q2 (Structural accuracy): There is a high structural similarity between the generated and reference RDSEFFs. The average Jaccard similarity, aggregated on 150 RDSEFFs, is 0.97 with 84.2% perfect matches. TeaStore requests are close to perfect: `StartPage` $J = 1.00$, `Login` $J = 0.95$, `CartAdd` $J = 0.95$, and `Logout` $J = 0.90$. We conclude that GARMA consistently captures the correct aggregation policy, presented in Section IV-C across different programming languages.

c) Q3 (CPU PMP accuracy): Table II shows the results for Q3. Overall, the CPU demand intervals and predicted means mostly align with measurements:

User-facing steps in `webui` and `auth` exhibit small absolute errors between means and high coverage (typically $>$ 85%), e.g., `webui:startPage` and `auth:isLoggedIn`. For example, `webui:cartAdd`, where the predicted average interval is sharp and still covers 90.1% of the measured points (see Table II). Lower coverage is observed on very cheap or database-dominated actions (e.g., `image:webImages`, `persistence:listCategories`, `persistence:findProduct`), where low CPU costs and external dependencies can make narrowing intervals harder for the LLM. For the endpoint `image:webImages`, we observed a slight overestimation (see Table II), resulting in a coverage of only 44.8%, but with the measured mean in the predicted interval. In general, operations that consume sub-millisecond CPU demand tend to yield less accurate estimates with GARMA. Another example of this is the `persistence:findProduct` endpoint that shows significant overestimation (predicted mean: 4.1 ms, measured: 0.30 ± 0.2 ms). The LLM estimating the PMPs could conflate waiting time for the external database call with CPU time of the persistence service itself. The actual persistence service performs minimal CPU work (a null check and copying ≈ 5 fields), but the LLM estimator allocates more processing time. Similar overestimation affects other persistence endpoints

with database calls (`persistence:listCategories`: predicted 0.92 ms vs. actual 0.50 ms).

Regarding interval width, GARMA generally produces comparatively sharp intervals that work well for stable CPU-bound behavior. For the TeaStore, sharpness is reliable for the `webui` and `auth` endpoints, whereas very small or database-driven actions cause inconsistent widths.

d) Comparison to the Naive Baseline LLM Estimator: Table II contrasts GARMA with the baseline LLM estimator. GARMA consistently yields lower absolute errors and more reliable coverage than the baseline.

GARMA provides accurate predicted means and stable interval widths for all CPU-relevant `webui` and `auth` actions, while the baseline overestimates many endpoints by large margins (e.g., `webui:startPage` and `auth:cartAdd`). The baseline performs better only on a few persistence endpoints, where GARMA overestimates due to an assumed conflated database wait time. A key difference is interval sharpness: GARMA maintains consistently narrow intervals for the main user-facing steps, whereas the baseline relies on wider bounds. Across all `webui` endpoints, GARMA averages 1.59 ms interval width vs. the baseline's 3.53 ms (2.2× sharper), with the largest gains on `webui:login` (4.0×) and `webui:cartAdd` (3.0×). Overall, GARMA yields more stable CPU-demand intervals than the baseline.

e) Q4 (Modeling time): We qualitatively assessed the architectural performance modeling and calibration effort. Previous work by Martens et al. [11] reports a total modeling time of approximately 200 min until a performance simulation with a PCM instance can be conducted. Of this, approximately 30 minutes are required to specify RDSEFFs when creating component-based performance models for a web server comparable to TeaStore. Defining PMPs takes 10 minutes per model, excluding the measurement experiments typically needed to parameterize these demands. Moreover, the work by Reussner et al. assumes familiarity with Palladio tooling [2]. In our workflow, RDSEFF construction and PMP calibration are fully automated and language agnostic. The workload dependency graph is derived from early design artifacts (e.g. scenario diagrams), while the runtime environment and workload characterization are provided as plain text. On average, 119 words for workload characterization and 112 words for runtime environment in our inputs. Even without a head-to-head time study, automating RDSEFF construction and PMP calibration, together with a short textual input, removes tasks that usually consume a substantial share of time.

Moreover, we systematically map GARMA's automation points to Smith's manual SPE process steps (Table III). GARMA assumes the early SPE steps are completed and uses the resulting execution-graph-like workload dependency graph plus workload/runtime text as inputs.

f) Answers to the GQM questions:

- **Q1** GARMA auto-fills 76.1 % of inputs for end-to-end execution, with human evaluation confirming most pre-filled entries are correct.

TABLE III

MAPPING OF SMITH'S SPE STEPS TO MANUAL ACTIVITIES VS. GARMA.

SPE	Manual	GARMA
1–4	Define scope & risk; identify critical use cases; select key scenarios & workload intensity; set performance objectives	Assumed done
5	Construct the performance model	**Input**: workload dependency graph, **Output**: RDSEFFs
6	Annotate execution steps with software-resource counts	Implicit via action-level RDSEFF + parametric demand terms (no explicit resource-count artifact)
7	Add computer resource requirements	**Input**: runtime-environment
7	Estimate demands (best/worst)	**Automates**: bounded PMP estimation
8	Evaluate models	**Output**: Valid RDSEFFs for simulation
9	Verify/validate models	**Automates**: PMP consistency checks + feedback loop

- **Q2** Generated RDSEFF structures closely match references (**0.97** average Jaccard; **84.2%** perfect).
- **Q3** CPU PMP intervals are decision-useful: TeaStore shows high coverage and sharpness when CPU costs dominate, and lower coverage for very small or DB-driven steps.
- **Q4** GARMA uses concise textual input to automate RDSEFF construction and PMP calibration, conceptually simplifying activities that prior work identified as a large share of modeling time and reduces tooling requirements.

GARMA is thus able to assess the RDSEFF structure almost correctly and to capture the main CPU costs without a running system and measurements.

VI. THREATS TO VALIDITY

This section discusses validity threats following Runeson, Höst and Sallou [26], [27].

Construct validity may be affected by prompt engineering bias, and metric selection bias. Prompt engineering bias is reduced through systematically designed prompts grounded in established software performance engineering principles. Metric selection bias is addressed by combining structural similarity with quantitative CPU-demand accuracy.

Internal validity threats stem from our measurement-based ground truth. CPU times for TeaStore may be influenced by instrumentation overhead, system noise, and environment variability. We mitigate this via multiple runs, outlier filtering, and documentation. Because GARMA relies on LLMs, their nondeterminism can introduce randomness. We mitigate this by a fixed LLM configuration and by producing multiple independent results for the same operation. Furthermore, we compare GARMA against an LLM baseline, which performs worse. Internal validity is also influenced by the accuracy of architect-provided early-design artifacts. Since architects typically define service boundaries, API endpoints, and cross-service interactions at design time, the required information is generally available, but inaccuracies can propagate into RDSEFFs and PMPs.

External validity is limited by the size of evaluated systems and by the focus on CPU demand. Other resource dimensions

(e.g., latency) were not assessed. Moreover, only GPT-5 was evaluated. To support comparative studies, we release prompts, responses, and datasets [20].

VII. Conclusion

This paper introduced an LLM workflow to generate architectural performance models directly from early-stage software artifacts. GARMA normalizes heterogeneous inputs, derives RDSEFF structures, and estimates bounded CPU resource demands *without requiring a running system*. GARMA produced performance models with high structural fidelity and meaningful CPU demand intervals. The autogenerated RDSEFFs closely matched manually created reference models (average Jaccard similarity 0.97), and the predicted CPU bounds aligned with empirical measurements for most operations, particularly for user-facing endpoints.

A single-pass LLM baseline often underestimates CPU demands in lightweight TeaStore actions, which leads to lower coverage and higher errors. GARMA in contrast provides more reliable interval predictions.

While our evaluation demonstrates the feasibility of LLM-assisted architectural performance modeling, more work is required to broaden applicability across different microservice applications, to improve the estimation of very small resource demands, and to enhance the interpretability of prediction intervals. Future work might extend GARMA.

Explainability: Enrich the generated PMPs with explanatory rationales to increase transparency and support architects in understanding the underlying assumptions.

Human-in-the-loop evaluation: Run a user study comparing human-created and LLM-created RDSEFFs, to assess differences in structure, parameterization, and modeling time.

Data Availability Statement

We provide a replication package for GARMA [20].

Acknowledgments

This work was funded by Core Informatics at KIT (KiKIT) of the Helmholtz Assoc. (HGF), the German Research Foundation (DFG) - SFB 1608 - 501798263, the Topic Engineering Secure Systems of the Helmholtz Association (HGF), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the National Research Data Infrastructure – NFDI 52/1 – 501930651 and supported by KASTEL Security Research Labs, Karlsruhe. This research was edited by our textician, Daniel Shea.

References

[1] S. J. Fowler, *Production-ready microservices: building standardized systems across an engineering organization*. "O'Reilly Media, Inc.", 2016.

[2] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, and A. Koziolek, *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016.

[3] C. U. Smith, "Origins of software performance engineering: Highlights and outstanding problems," in *International Workshop on Software and Performances*. Springer, 2000, pp. 96–118.

[4] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE '07)*. IEEE, 2007, pp. 171–187.

[5] S. Spinner, G. Casale, F. Brosig, and S. Kounev, "Evaluating approaches to resource demand estimation," *Performance Evaluation*, 2015.

[6] J. Grohmann, N. Herbst, S. Spinner, and S. Kounev, "Self-tuning resource demand estimation," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 21–26.

[7] K. Krogmann, M. Kuperberg, and R. Reussner, "Using genetic search for reverse engineering of parametric behavior models for performance prediction," *IEEE Transactions on Software Engineering*, 2010.

[8] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd ACM/SPEC ICPE*. ACM, 2012, pp. 247–248.

[9] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora, "A framework for measurement based performance modeling," in *Proceedings of the 7th International Workshop on Software and Performance*, 2008, pp. 55–66.

[10] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tůma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, 2019.

[11] A. Martens, S. Becker, H. Koziolek, and R. Reussner, "An empirical investigation of the effort of creating reusable, component-based models for performance prediction," in *International Symposium on Component-Based Software Engineering*. Springer, 2008, pp. 16–31.

[12] C. U. Smith and L. G. Williams, *Performance solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley Reading, 2002, vol. 1.

[13] R. Jain, *The art of computer systems performance analysis*. john wiley & sons, 1990.

[14] C. U. Smith, "Software performance engineering," in *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*. Springer, 1993, pp. 509–536.

[15] T. Zheng, C. M. Woodside, and M. Litoiu, "Performance model estimation and tracking using optimal filters," *IEEE Transactions on software engineering*, vol. 34, no. 3, pp. 391–406, 2008.

[16] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance evaluation*, vol. 67, no. 8, 2010.

[17] T. Kappler, H. Koziolek, K. Krogmann, and R. Reussner, "Towards automatic construction of reusable prediction models for component-based performance engineering," in *Software Engineering 2008*. Gesellschaft für Informatik e. V., 2008, pp. 140–154.

[18] G. Garbi, E. Incerto, and M. Tribastone, "Learning queuing networks by recurrent neural networks," in *Proceedings of the ACM/SPEC international conference on performance engineering*. ACM, 2020.

[19] M. Hummel, N. Hagel, M. Kaushik, J. Keim, E. Burger, and H. Koziolek, "Llm-assisted microservice performance modeling," in *16th Symposium on Software Performance*, 2025.

[20] M. Hummel, N. Hagel, D. Fuchss, S. Corallo, M. Kaushik, J. Keim, H. Koziolek, and R. Reussner, "Replication Package: GARMA: Generative Architectural Resource Demand Estimation for Microservice Applications," 2026.

[21] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 728–738, 1984.

[22] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, pp. 528–532, 1994.

[23] P. Jaccard, "Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines," *Bull Soc Vaudoise Sci Nat*, 1901.

[24] M. Mazkatli, D. Monschein, M. Armbruster, R. Heinrich, and A. Koziolek, "Continuous integration of architectural performance models with parametric dependencies–the cipm approach," *Automated Software Engineering*, vol. 32, no. 2, p. 54, 2025.

[25] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *IEEE MASCOTS*. IEEE, 2018, pp. 223–236.

[26] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[27] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.