AutoPCM: Automated Generation of Architectural Performance Models from Early Design Artifacts using LLM Agents

Maximilian Hummel[1,2][0009-0001-4123-2786],
Dominik Fuchß[2][0000-0001-6410-6769], Sophie Corallo[2][0000-0002-1531-2977],
Nathan Hagel[2][0009-0003-9919-4449], Jan Keim[2][0000-0002-8899-7081],
Heiko Koziolek[1][0000-0002-8805-6206], and Ralf Reussner[2][0000-0002-9308-6290]

[1] ABB Corporate Research Center, Mannheim, Germany
{heiko.koziolek, maximilian.hummel}@de.abb.com
[2] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{dominik.fuchss,sophie.corallo,nathan.hagel,jan.keim,ralf.reussner}@kit.edu

Abstract. Architectural performance models, such as the Palladio Component Model (PCM), allow architects to evaluate design alternatives for software systems through performance simulation. The dominant barrier to such early-design simulation, however, is the construction of a valid performance model in the first place: building a complete PCM instance requires considerable performance engineering expertise and manual effort, and a metamodel-valid instance is a necessary precondition for any subsequent simulation. Existing approaches to (semi-) automatically construct such models either require a running system or its source code, depend on mature annotated UML models, or produce only partial models that still need manual completion. We present AutoPCM, an LLM-based multi-agent workflow that processes heterogeneous early-design artifacts and automatically constructs complete, metamodel-valid PCM instances. The conceptual contribution is a role-based agentic decomposition that generalizes to other component-based performance formalisms (e.g., Layered Queueing Networks); in this paper, we instantiate and evaluate it for PCM using GPT-5.4. In an evaluation on two subject systems, Corona-Warn-App (CWA) (11 use cases) and TeaStore (9 use cases), 205/220 generated PCM instances pass PCM metamodel validation. The generated models achieve a mean structural similarity of 0.86 (Jaccard) compared to manually authored reference models.

Keywords: Automated Performance Modeling · Palladio Component Model · LLM.

1 Introduction

Exploration of design alternatives before implementation and deployment is enabled by architectural performance models, such as the Palladio Component Model (PCM) [22]. Architects can use PCM to simulate the timing behavior of

a system. The major barrier to such early-design performance simulation, however, is the construction of a performance model: the necessary precondition for any subsequent simulation is a metamodel-valid PCM instance, and constructing one requires performance expertise [25, 28] as well as substantial manual effort. For example, Martens et al. [20] assume familiarity with the Palladio tooling, and still they report approximately 200 minutes of modeling time for a single web server scenario. The barrier of performance model construction is explained by the fact that modeling creates components, behavioral specifications with resource demands, component assemblies, hardware descriptions, and user workload characteristics. During early design, however, such information is often scattered across informal artifacts such as diagrams and sketches. Consequently, architects often lack the information needed to characterize performance trade-offs upfront but still must commit to wide-ranging architectural decisions.

Existing approaches to reduce manual labor for constructing performance models either depend on detailed annotated UML models [2, 19, 29] or require a running system and its source code [5, 14, 17, 18]. Neither is available during early design. Recently, LLM-based approaches have begun tackling parts of the problem, for instance, by generating architecture models from informal specifications [27] or by estimating resource demands from early-design artifacts [11, 12]. These approaches yield only partial models, either structural models lacking performance annotations or resource demands requiring still manual integration into a performance model. To date, no approach generates a complete, metamodel-valid performance model from informal artifacts available during early design.

This paper introduces AutoPCM, an LLM-based multi-agent workflow that processes heterogeneous early-design artifacts and automatically constructs complete, metamodel-valid PCM instances. AutoPCM requires a set of artifacts as input, which are typically available in early design [26], such as informal sequence diagrams, architecture descriptions, deployment specifications, and load profiles. AutoPCM automates the subsequent formal performance model construction and validation steps. AutoPCM consists of two phases: (i) a pre-processing phase in which a Retriever Agent extracts a structured knowledge base from all input documents, and (ii) an agent-based model construction phase in which four specialized LLM agents (i.e., Component Developer, Software Architect, System Deployer, and Domain Expert) populate a PCM view following the role-based PCM process model by Koziolek et al. [16]. A deterministic cross-model validator then checks the assembled PCM model instance for structural validity, cross-view consistency, and field-level completeness.

We tested AutoPCM on two systems using PCM as the target metamodel and GPT-5.4 as the underlying LLM: TeaStore (an academic reference system with 9 use cases) and Corona-Warn-App (a commercial system with 11 use cases). 205 of 220 generated PCM instances passed PCM metamodel validation. The generated models achieved a mean structural similarity of 0.86 (Jaccard coefficient) compared to manually authored reference models. While we instantiate AutoPCM for PCM, the underlying role-based multi-agent decomposition

is independent of PCM and transferable to other component-based performance formalisms (e.g., Layered Queueing Networks).

2 Background

The *Palladio Component Model (PCM)* [22] is an architecture-level modeling approach allowing software architects to predict quality attributes of component-based systems, including response time, throughput, and resource utilization, before implementation. In this work, components correspond to microservices. To illustrate each view concretely, we use two microservices from the TeaStore microservice application [15] as a running example: *WebUI*, which serves the storefront, and *Auth*, which manages authentication. A PCM instance is structured into five architectural views, each capturing a distinct concern of the system. Koziolek et al. [16] define four developer roles that are each responsible for constructing one or more of these views: the Component Developer specifies the repository model (components, interfaces, and RDSEFFs), the Software Architect assembles the system model, the System Deployer defines the resource environment and allocation, and the Domain Expert describes the usage model. Once all views are populated, the resulting model can be simulated using tools such as SimuLizar to obtain performance predictions under various workload and deployment configurations. The *repository model* defines the software building blocks: components with provided and required interfaces and, for each operation, a behavioral specification (called *RDSEFF*) that abstracts the operation's control flow into local computations (internal actions) annotated with resource demands and calls to other components (external call actions, e.g., WebUI provides a `startPage` operation whose RDSEFF contains an external call to Auth's `isLoggedIn`). The *system model* describes how component instances are assembled and wired together to form a running system, including the entry points (system calls) exposed to external users (e.g., WebUI and Auth are each instantiated as an AssemblyContext, connected via a connector). The *resource environment* captures the hardware infrastructure as computational nodes with their processing capacities and the network links between them (e.g., two processing nodes hosting WebUI and Auth). The *allocation model* maps each component instance to exactly one particular infrastructure node (e.g., WebUI allocated to one node, Auth to another). Finally, the *usage model* specifies how users interact with the system: which operations they invoke, in what order, and at what frequency (e.g., an open workload with a given inter-arrival time issuing a system call to WebUI's `startPage`, or a closed workload with a fixed user population).

3 Related Work

A variety of approaches exist to generate architecture and performance models from development artifacts. Woodside et al. [29] generated Layered Queueing Networks (LQNs) and Petri Nets from annotated UML diagrams. Li et al. [19] and Altamimi et al. [2] follow a similar idea of transforming annotated UML

models into LQNs. The above-mentioned approaches rely on mature and correctly annotated UML models, which, especially during early design phases, are not always feasible.

Kappler et al. [14] generated performance models, namely Palladio Component Model (PCM) instance from Java code. Langhammer et al. [18] extract performance models using the system's source code and test cases. Other works use monitoring data or measurements to automatically extract [5] or further improve performance models [21], including model-driven continuous performance engineering for microservice-based systems [7]. All these cases require a running system and its source code to extract or generate performance models. A recent systematic mapping by Eramo et al. [8], covering 66 primary studies, identifies automated generation/extraction of architectural performance models as a comparatively under-investigated area and highlights the field's strong reliance on runtime-monitored data.

With the ever-improving capabilities of Large Language Models (LLMs), their adoption for software architecture tasks is increasing [9]. As LLMs output textual artifacts, many approaches, like [27, 30] make use of textual Domain-Specific Languages (DSLs), which can later be further processed or transformed into more formal architectural or performance models. Tagliaferro et al. [27] use LLMs to generate UML component diagrams. Adnan et al. [1] evaluated LLM agents for autonomous microservice generation, but target code rather than architectural performance models. While there is already widespread adoption of generating models from a single source of truth, Esposito et al. [9] identify the consideration of heterogeneous input artifacts as an important research gap and future direction in GenAI-supported software architecture. Related works for adopting LLMs to generate architectural performance models like the PCM is limited. Hummel et al. [11, 12] make use of LLMs to generate one aspect of a performance model, namely SEFFs and estimated resource demands. While heterogeneous artifacts are leveraged to improve resource demand estimations using LLMs, manual modeling is still required to make use of resource demands (i.e., using them in a performance simulation). The present work addresses this gap by generating functional architectural performance models from a set of heterogeneous artifacts.

4 Method

The goal of AutoPCM is to transform various early-design artifacts (e.g., diagrams, textual descriptions, source code) into a complete, metamodel-valid architectural performance model which thereby constitutes a usable input for downstream performance simulation. AutoPCM targets models that decompose a system into complementary views: components and their behavioral specifications, system model, deployment infrastructure, and user workload. We instantiate AutoPCM for the Palladio Component Model (PCM) [22], but the pipeline structure generalizes to other performance modeling formalisms (e.g., Layered Queueing Networks). As shown in Figure 1, the processing pipeline of AutoPCM

consists of two phases: (1) a *Pre-Processing* phase that extracts a structured Global System Knowledge (GSK) from the input documents (Section 4.2), and (2) an *Agent-based Model Construction* phase in which four specialized LLM agents each populate one or more model views (Section 4.3). Both phases share a target metamodel (PCM in our case) acting as a blackboard: each agent in AutoPCM reads the output of its predecessors and writes its own model view directly into this blackboard. (To support replication and comparative studies, we release all prompts, agent traces, generated models, and reference models [10].)

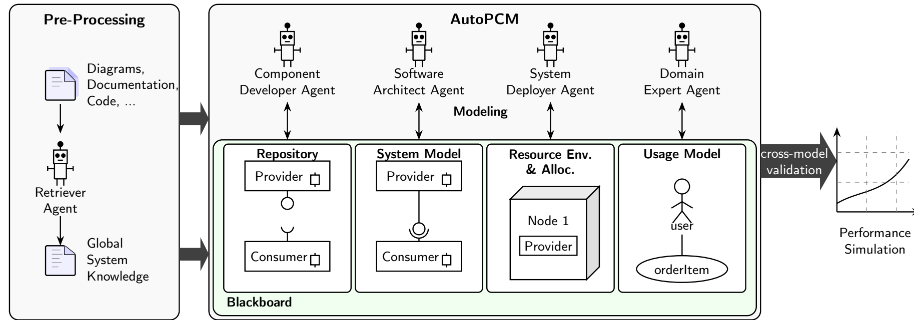

Fig. 1. Overview of the AutoPCM Agentic Pipeline

4.1 Inputs

AutoPCM can process a set of various artifacts as input. AutoPCM requires no standard input notation and so is broadly applicable because architects can use their own notations or conventions.

Documents may be, for example, scenario diagrams, textual descriptions, component graphs, or source code. Diagrams are represented in text-based notations (e.g., PlantUML, SVG markup) and processed by the LLM as structured text. Modern multimodal LLMs are capable of accepting images (e.g., a photo of a whiteboard drawing) as input, broadening the set of acceptable artifact formats. Each artifact is represented as a triple `(type, id, content)`. The `type` is drawn from a fixed vocabulary of seven categories: `sequence_diagram`, `architecture_diagram`, `api_specification`, `source_code`, `deployment_manifest`, `deployment_config`, and `load_profile`. The type label is assigned automatically by the Retriever Agent during pre-processing (Section 4.2), removing the need for manual annotation. This vocabulary enables deterministic artifact-to-agent assignment in the downstream model construction phase: each role agent declares the artifact types it consumes (e.g., the System Deployer receives only `deployment_manifest` and `deployment_config` artifacts), ensuring that agents operate on a focused, role-relevant subset while the full context remains available through the GSK (Section 4.2).

4.2 Pre-Processing: Knowledge-Base Extraction

The first phase of the processing pipeline is carried out by a Retriever Agent, a ReAct [31] tool-calling agent that explores the document collection and produces a *Global System Knowledge (GSK)*. ReAct agents interleave reasoning traces with tool calls, allowing the LLM to plan its next action based on observations from previous steps. The agent operates on the full artifact set without chunking or summarization. The used LLM (GPT-5.4) provides a context window of approximately one million tokens, corresponding to roughly 3,000 pages of plain text. In practice, the effective capacity is lower because instruction-following quality and factual accuracy degrade with context size. In our case, each tested scenario comprises four artifacts totaling approximately 530–980 tokens of structured text (component diagram, sequence diagram, deployment configurations, and load profiles). Aggregated over all use cases, the full artifact set remains below 5,500 tokens for TeaStore (9 scenarios) and below 10,000 tokens for CWA (11 scenarios), fitting well within the model's context capacity. Even the complete CWA project documentation (41 files, approximately 87,000 tokens using the cl100k_base tokenizer) would still fit within the context window of current LLMs with one million tokens. For input sets that exceed the context window, a prior segmentation step or retrieval-augmented strategy would be required. We leave such scalability extensions to future work. As a first step, the Retriever classifies each artifact into one of the seven type categories using an LLM-based classifier, enabling the downstream deterministic artifact-to-agent routing. It then retrieves a metadata index of all available artifacts (type, identifier, size) and reads each artifact individually through successive tool calls, reasoning about its content.

Global System Knowledge (GSK). The Retriever synthesizes a structured Markdown knowledge base from all input documents. It is organized into eight sections aligned with the target performance metamodel (PCM), covering system structure, interfaces, behavioral flows, implementation details, workload characteristics, infrastructure, and assembly configuration. Every extracted fact carries an inline provenance tag `[artifact_id]` that traces it back to its source document. The GSK serves as shared global context for all downstream agents. Since the extraction is performed by an LLM, the GSK may in principle omit or misrepresent information. Three mechanisms mitigate this risk: (i) the mandatory structure embedded in the retriever prompt ensures that no knowledge category is skipped entirely. (ii) provenance tags make every extracted fact traceable, so omissions and errors are auditable against the source artifacts; and (iii) downstream role agents are not limited to the GSK. They also read their assigned raw artifacts directly (cf. Section 4.3), so the GSK acts as a shared summary layer rather than an exclusive information bottleneck.

4.3 AutoPCM: Role-Based PCM Model Construction

The second phase of the pipeline (right side of Figure 1) employs four specialized ReAct LLM agents, each corresponding to a performance modeling role. We

follow the role taxonomy and view assignment by Koziolek et al. [16] (described in Section 2). The agents execute in a dependency-aware order: Component Developer, Software Architect, System Deployer, and Domain Expert. Each agent (i) reads the GSK, (ii) reads its assigned artifact subset, (iii) receives a structured summary of predecessor agents' output from the blackboard, and (iv) writes its model view directly into the blackboard. Crucially, no single agent ingests all inputs at once: the artifact-to-role assignment (described in Section 4.1) ensures that each agent's artifacts are limited to the artifact types relevant to its role (e.g. the System Deployer receives only `deployment_manifest` and `deployment_config` artifacts). The GSK and assigned artifacts are read on demand by the ReAct agents through individual tool calls and are not put into the prompt. Predecessor output is injected into the prompt, but only as a compact summary (component names, operation signatures, assembly context names, connector edges) rather than the full predecessor state. All role agents share three common tools:

– `list_artifacts`: Returns metadata only
– `read_artifact`: Returns the content of a single artifact
– `read_gsk`: Returns the GSK

Moreover, each agent adds role-specific `save` tools that write its model view to the blackboard, e.g., `save_components` and `save_seff` for the Component Developer, or `save_system` for the Software Architect. Figure 2 illustrates the internal structure of the Software Architect Agent.

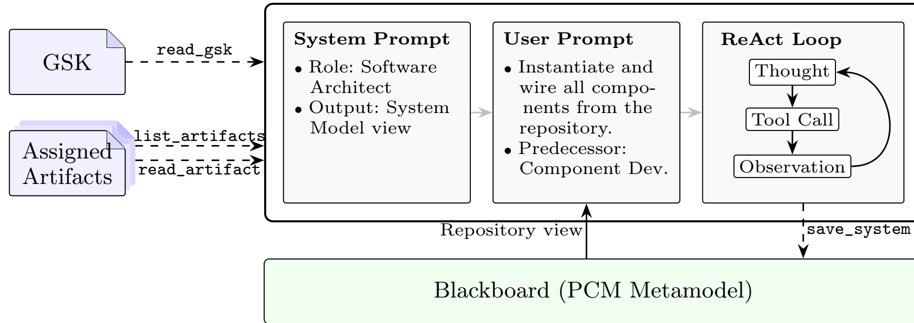

Fig. 2. Internal structure of the Software Architect Agent. Dashed arrows denote tool calls during the ReAct loop; solid arrows denote prompt-injected context

Component Developer: Repository Model. This agent identifies the system's components (e.g., microservices), each describing a reusable building block that may be instantiated multiple times in the system model. It models the provided and required interfaces as well as behavioral specifications for each operation. In PCM, these specifications are realized as Resource Demanding Service Effect

Specifications (RDSEFFs), enriched with best-case/worst-case resource demand estimations following Hummel et al. [11]. Behavioral specification construction follows an aggregation policy designed to keep the simulation model manageable [21]: local computations become internal actions with CPU resource demands, calls to other services become external call actions. Control-flow constructs (i.e., branches, loops) are modeled only when they contain external call actions; else, they are aggregated into a single internal action. Leaf services with no outgoing calls collapse to minimal RDSEFFs: Start, internal action, Stop. For example, the Component Developer identifies WebUI and Auth as components and models the `startPage` RDSEFF with an external call to Auth's `isLoggedIn`.

Software Architect: System Model. This agent (see Figure 2) constructs the system model by instantiating components and wiring their interactions. It creates component instances (AssemblyContexts in PCM) from the component types defined in the repository. The agent receives the component type list and the external call graph extracted from the behavioral specifications produced by the Component Developer. The architect agent creates an instance for each component, a connector wiring a required role to a provided role for each inter-component dependency, and it identifies system entry points exposed to users. For example, this yields two AssemblyContexts (WebUI, Auth), a connector wiring them, and an entry point on WebUI's `startPage`.

System Deployer & Domain Expert The remaining two agents follow the same ReAct pattern and tool structure, each populating the model views assigned to its role. The System Deployer constructs the Resource Environment and Allocation views, while the Domain Expert constructs the Usage Model. For example, the Domain Expert models the start page scenario as an open workload with a system call to WebUI's `startPage`.

4.4 Cross-Model Validation

After all four agents complete, a deterministic validator checks the assembled model along three dimensions: *(A) Structural validity*, well-formedness within each view (e.g., every usage scenario has exactly one start and one stop action); *(B) Cross-consistency*, that every inter-view reference points to a declared element (e.g., a system call in the usage model targets an operation declared in the repository); *(C) Completeness*, a field-level coverage metric that lists model fields left unpopulated, surfacing missing inputs (e.g., an absent `load_profile`) as auditable information gaps and quantifying residual uncertainty.

4.5 Output: Simulatable PCM Instance

AutoPCM produces a PCM instance covering all architectural views: the Repository Model, the System Model, the Resource Environment, the Allocation, and the Usage Model. Figure 3 shows a generated TeaStore PCM instance loaded in Palladio, where the instance can be inspected and adapted for further what-if

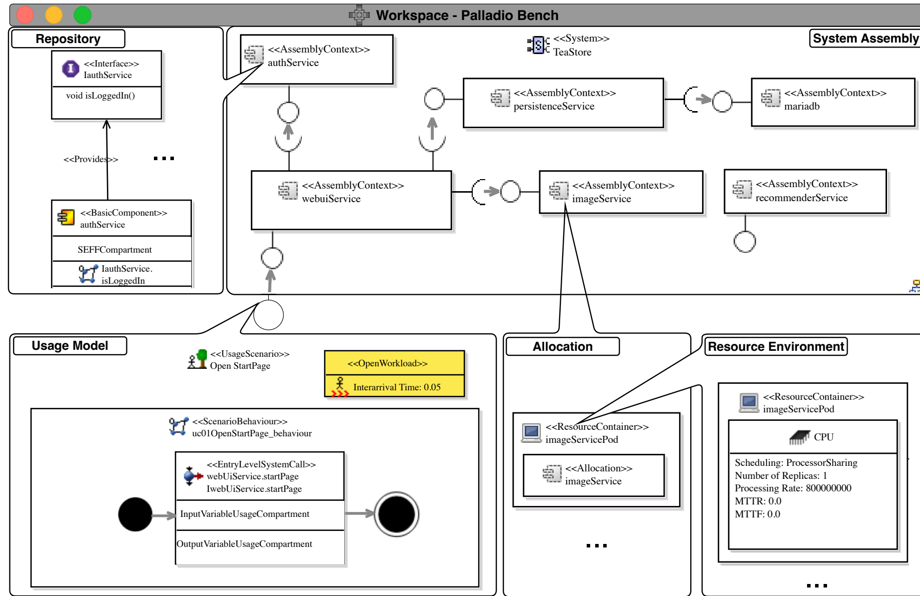

Fig. 3. Generated PCM instance for the TeaStore `StartPage` use case

analyses. Every extracted value includes a provenance tag referencing its source artifact. The output is accompanied by intermediate post-agent-step snapshots for purposes of traceability. Once metamodel-valid, the instance is a usable input for PCM simulators like SimuLizar or SimuCom; these simulators use the control flow structures and resource demands of the system to simulate the timing behavior and in this way, are able to yield response time, throughput, and resource utilization predictions. Architects can modify the model (e.g., the user workload or integration of another component) and then re-simulate to assess the impact on overall system performance. AutoPCM establishes metamodel validity as necessary precondition; however, the *accuracy* of the resulting simulation predictions is outside the scope of this paper.

5 Evaluation

We evaluate AutoPCM following the Goal-Question-Metric (GQM) plan by Basili et al. [3, 4].

Goal: Analyze PCM instances produced by AutoPCM from heterogeneous early-design artifacts to assess their suitability as starting models for performance simulation, with respect to metamodel conformance, structural correctness, and modeling effort, from the viewpoint of a software performance engineer. We refine this high-level goal into these three questions:

Q1 (Metamodel validity). *Are the PCM instances generated by AutoPCM metamodel-valid according to the PCM Ecore metamodel and its OCL constraints?*

Metamodel validity is a necessary precondition for simulation: only valid instances are loadable by Palladio tooling and can be simulated. We validate each generated model against the PCM Ecore metamodel using the Eclipse Modeling Framework (EMF) validation tooling. Each run yields a binary pass/fail verdict. We report the Ecore conformance rate as the percentage of runs that produce XMI files loadable without validation errors.

Q2 (Structural correctness). *How structurally correct are the generated PCM instances compared to manually authored reference models?* We measure structural correctness using the Jaccard coefficient [13], which quantifies set similarity as the ratio of the intersection to the union of two sets. We treat every model element in a generated PCM instance and its manually authored reference as set members; two elements are considered equal when their names, referenced elements, and structural properties match. By incorporating the references to other model elements, this encoding makes Jaccard sensitive to topology. We report the coefficient per view (e.g., JC_{repo}) and globally (JC_{global}), where the global coefficient treats all elements across all views as a single set.

Q3 (Modeling effort). *How much modeling effort is required to obtain performance models using AutoPCM compared to manual performance modeling?* We qualitatively assess the reduction in manual effort by comparing against a prior empirical study on manual PCM construction.

5.1 Subject Systems

We evaluate AutoPCM on two subject systems of varying domain and complexity. The **TeaStore** [15] is a Java-based microservice reference application for benchmarking. We evaluate nine use cases, covering start page access, login, category browsing, product viewing, cart operations, product ordering, profile viewing, and logout. The **Corona-Warn-App (CWA)** [6] is a large-scale mobile health application with a complex backend architecture. We evaluate eleven use cases covering onboarding, diagnosis key submission, and data donation workflows. Input artifacts per use case include a component diagram (PlantUML), sequence diagram (PlantUML), deployment configurations (text), and load profiles (text). Reference PCM models are manually authored. We reviewed the CWA use cases in a one-hour review session with four software architecture researchers not involved in the study's design or implementation [10].

Experimental setup. We use GPT-5.4 via Azure OpenAI with temperature 0. For each use case, we generate $N = 10$ model instances to capture residual LLM nondeterminism. In addition, we run each subject system with all use case artifacts combined to assess whether the approach handles larger, cross-cutting inputs with redundant architectural elements across use cases.

5.2 Results

Q1 (Metamodel validity). The PCM conformance results are shown in the "Valid" column of Table 1. Of the 220 generated PCM instances (22 scenarios $\times$ 10 runs,

including one combined-input run per subject system), 205 pass the PCM validation across all views, yielding a conformance rate of 93.2%. All TeaStore instances (100/100) are valid, including the combined-input run. Figure 3 shows a valid generated PCM instance loaded into the Palladio bench. For CWA, 105 of 120 instances pass, concentrating the 15 failures in CWA use cases. The dominant failure pattern is an invalid usage model: the Domain Expert Agent generates system calls that reference operations not defined in the component specification produced by the Component Developer. For example, in CWA UC02, the LLM generates a call to `portalServer.generateTeletan` although the predecessor context lists `portalServer` with no operations. This occurs because the LLM's domain knowledge about the CWA system overrides the explicit structural constraint in the prompt to not produce system calls for non-existing operations. The "All UCs – combined" rows in Table 1 report validity when all use case artifacts are provided to AutoPCM at once. Despite redundancy across use cases, AutoPCM yields valid PCM instances in 10/10 runs for TeaStore and 8/10 for CWA, producing larger models (287 and 374 elements) and confirming scalability to broader architectural slices without loss of validity.

Q2 (Structural correctness). Table 1 presents the per-view and global Jaccard coefficients for all scenarios. Since the usage model describes system *use* rather than the system itself, JC_{usage} is reported only in the replication package [10]. Across all 20 scenarios, AutoPCM achieves an aggregate JC_{global} of 0.86, meaning that on average approximately 86% of model elements in the generated instances match the manually authored reference. TeaStore achieves a higher mean global Jaccard (≈0.89) than CWA (≈0.83), reflecting that TeaStore's explicit REST-based service boundaries map more directly to architectural model elements than CWA's mobile clients, peer-to-peer protocols, and larger, use-case-specific component subsets. Across both systems, the allocation view ($JC_{\text{alloc}} = 0.92$) scores highest. The repository ($JC_{\text{repo}} = 0.87$) is strong overall, with accuracy correlating with the clarity of inter-service interactions in the input artifacts. The system view ($JC_{\text{sys}} = 0.81$) is weakest, primarily because the approach generates connectors for all components for which architectural evidence exists, including those not exercised in the evaluated use case. More broadly, the dominant pattern behind remaining deviations is over-generation of model elements for which the input artifacts provide insufficient evidence. The approach generates such elements rather than omitting them, which broadens architectural coverage but means those elements have limited evidential support. Overall, all views consistently achieve Jaccard coefficients above 0.8 in aggregate, indicating that the generated models are structurally very similar to the manually authored references across both subject systems. Run-to-run variance is low across all scenarios (std of $JC_{\text{global}} \leq 0.08$), indicating that AutoPCM produces stable results despite LLM nondeterminism; only a few mildly elevated per-view deviations ($\approx$ 0.13–0.15) appear in the system and repository views, where connector and component-set choices vary most.

Q3 (Modeling effort). Prior work by Martens et al. [20] reports approximately 200 minutes of total modeling time as a manual baseline until a PCM instance

Table 1. Structural correctness of generated PCM instances: Jaccard reported as mean ± std. #Elem. denotes the total number of model elements in a use case.

	Use Case	#Elem.	JC_{repo}	JC_{sys}	JC_{res}	JC_{alloc}	JC_{global}	Valid
TeaStore	UC01 – StartPage	70	0.91 ± 0.10	0.85 ± 0.09	0.90 ± 0.05	0.92 ± 0.09	0.89 ± 0.04	10/10
	UC02 – Login	64	0.92 ± 0.10	0.84 ± 0.15	0.89 ± 0.04	0.93 ± 0.09	0.91 ± 0.07	10/10
	UC03 – BrowseCategory	97	0.90 ± 0.08	0.76 ± 0.10	0.89 ± 0.09	0.82 ± 0.04	0.87 ± 0.06	10/10
	UC04 – ViewProduct	100	0.89 ± 0.06	0.83 ± 0.04	0.89 ± 0.05	0.95 ± 0.08	0.90 ± 0.05	10/10
	UC05 – AddToCart	66	0.96 ± 0.08	0.85 ± 0.14	0.89 ± 0.04	0.97 ± 0.07	0.89 ± 0.06	10/10
	UC06 – ViewCart	100	0.93 ± 0.06	0.86 ± 0.04	0.90 ± 0.00	0.97 ± 0.07	0.93 ± 0.02	10/10
	UC07 – PlaceOrder	74	0.85 ± 0.14	0.87 ± 0.15	0.90 ± 0.00	0.92 ± 0.09	0.89 ± 0.08	10/10
	UC08 – ViewProfile	89	0.90 ± 0.04	0.82 ± 0.04	0.90 ± 0.00	0.85 ± 0.05	0.89 ± 0.02	10/10
	UC09 – Logout	49	0.89 ± 0.04	0.82 ± 0.13	0.84 ± 0.08	0.85 ± 0.05	0.87 ± 0.04	10/10
	All UCs – combined	287	–	–	–	–	–	10/10
CWA	UC01 – Onboarding	79	0.84 ± 0.14	0.78 ± 0.08	0.91 ± 0.10	0.93 ± 0.10	0.84 ± 0.08	10/10
	UC02 – BackgroundTracing	89	0.88 ± 0.11	0.76 ± 0.06	0.90 ± 0.06	0.97 ± 0.05	0.79 ± 0.05	2/10
	UC03 – RegisterPCRTest	83	0.75 ± 0.13	0.75 ± 0.07	0.84 ± 0.07	0.89 ± 0.07	0.81 ± 0.07	9/10
	UC04 – RetrieveTestResult	106	0.73 ± 0.07	0.73 ± 0.07	0.86 ± 0.06	0.89 ± 0.05	0.74 ± 0.06	9/10
	UC05 – ShareDiagnosisKeys	113	0.89 ± 0.08	0.91 ± 0.07	0.93 ± 0.07	0.93 ± 0.08	0.91 ± 0.06	10/10
	UC06 – ExposureWarning	108	0.80 ± 0.08	0.85 ± 0.03	0.96 ± 0.04	0.98 ± 0.04	0.85 ± 0.04	10/10
	UC07 – TeleTANVerification	132	0.89 ± 0.06	0.90 ± 0.05	0.95 ± 0.04	0.98 ± 0.04	0.87 ± 0.04	10/10
	UC08 – EventCheckin	76	0.93 ± 0.08	0.76 ± 0.02	0.89 ± 0.10	0.91 ± 0.10	0.90 ± 0.06	10/10
	UC09 – EventWarningUpload	104	0.80 ± 0.11	0.81 ± 0.08	0.91 ± 0.09	0.94 ± 0.09	0.89 ± 0.07	10/10
	UC10 – RapidAntigenTest	122	0.66 ± 0.10	0.67 ± 0.05	0.73 ± 0.08	0.75 ± 0.07	0.65 ± 0.06	9/10
	UC11 – DataDonation	102	0.98 ± 0.03	0.85 ± 0.09	0.94 ± 0.07	0.97 ± 0.04	0.92 ± 0.06	8/10
	All UCs – combined	374	–	–	–	–	–	8/10
	Aggregate	91	0.87	0.81	0.89	0.92	0.86	205/220

for a web server is ready for performance simulation. Moreover, this assumes familiarity with the Palladio tooling [22]. With AutoPCM, the entire PCM construction across all views is fully automated. The user provides heterogeneous design artifacts as input that are available early in the software performance engineering process [26]. AutoPCM constructs the model without requiring Palladio expertise. Even without a controlled time study, automating the construction of all views from concise textual and diagrammatic input removes tasks that prior work identified as consuming a substantial share of modeling effort.

In Table 2, we map the automation scope of AutoPCM onto the Software Performance Engineering (SPE) process proposed by Smith [26]. AutoPCM assumes the early SPE steps (scope definition, scenario selection, workload characterization) are completed and uses the resulting artifacts as input. Next, AutoPCM automates the construction, annotation, and validation of the performance model.

Answers to the GQM questions.

Q1 93.2% of generated PCM instances (205/220) pass the PCM metamodel validation across all views; as a result, the structural precondition for downstream simulation is met. All TeaStore instances are valid; the cause of the 15 CWA failures is simply the fact that the LLM generates usage model references to operations which are, in fact, absent from the component specification. This

Table 2. Mapping of Smith's SPE process steps to manual activities vs. AutoPCM.

SPE Step	Manual	AutoPCM
1–4	Define scope & risk; identify critical use cases; select key scenarios & workload intensity; set performance objectives	Assumed completed (input prerequisite)
5–6	Construct the performance model from scenarios; annotate execution steps with resource counts	Automated: multi-agent PCM construction across all views; action-level SEFF construction with parametric demand terms, following Hummel et al. [11]
7	Add computer resource requirements	Input: runtime environment artifacts processed by System Deployer Agent
8	Evaluate models	Output: metamodel-valid PCM instances (input to simulators)
9	Verify/validate models	Automated: structural and cross-consistency validation

behavior of the LLM is attributable to residual nondeterminism where domain knowledge overrides structural constraints.

Q2 The generated models achieve a mean global Jaccard coefficient of 0.86 across all 20 scenarios (0.89 for TeaStore, 0.83 for CWA). All views consistently exceed 0.8 in aggregate; accuracy is highest for views derived from explicit artifacts (resource environment, allocation) and for scenarios with unambiguous inter-service interactions.

Q3 AutoPCM automates the construction of all views from heterogeneous input artifacts, removing tasks that a prior empirical study [20] identified as requiring substantial manual effort and Palladio expertise.

6 Threats to Validity

We discuss threats to validity following Runeson and Höst [23] but also following Sallou et al. [24] for LLM-specific threats. *Construct validity* may be affected by prompt engineering bias, reference model subjectivity, and metric selection bias. We reduce prompt bias through systematic prompts grounded in both software performance engineering principles and the PCM role-based process model [16]. To mitigate subjectivity in manually authored PCM reference models for structural correctness (Q2), the CWA models underwent review by four independent software architecture researchers. To make disagreements transparent, we report Jaccard coefficients per view. Metric selection bias is addressed by combining metamodel conformance (Q1), structural similarity via Jaccard coefficients (Q2), and qualitative effort comparison with manual modeling (Q3). However, these metrics do not assess simulated performance prediction accuracy. An OCL-valid PCM instance can still contain semantically problematic values (e.g., zero-probability branches) and structurally different models may yield identical simulations. Jaccard similarity is only a lower bound on semantic correctness; end-to-end simulation accuracy remains future work. *Internal validity* may be affected by the multi-agent pipeline architecture and LLM

nondeterminism. Because agents execute sequentially and each reads predecessor output from the blackboard, errors introduced by an early agent (e.g., the Component Developer) could propagate to downstream agents. The cross-model validation cannot detect semantically plausible yet incorrect model elements but it will catch referential and structural inconsistencies. Similarly, the GSK extracted by the Retriever Agent serves as shared context for all role agents; if it omits or misrepresents information, the omission may go undetected despite our three mitigation mechanisms described in Section 4.2. LLM nondeterminism introduces variance across runs. We mitigate this by generating ten instances per use case and reporting aggregated metrics and the variance. Another potential threat is the accuracy of the provided early-design artifacts. Since architects typically define service boundaries, API endpoints, and cross-service interactions at design time, the required information is normally available, but inaccuracies could propagate into the performance model. *External validity* is limited by the number and type of evaluated systems. Although TeaStore and CWA differ in complexity, domain, scale, and architectural style, both are backend-heavy service-oriented systems; applicability to other paradigms (e.g., event-driven) remains to be demonstrated. Moreover, only GPT-5.4 was evaluated. Different model families may exhibit other strengths or biases. Because AutoPCM currently processes all artifacts without chunking, scalability may be threatened for larger documentation sets.

7 Conclusion

We proposed AutoPCM, an LLM-based multi-agent workflow that constructs complete, metamodel-valid architectural performance models (PCM) from heterogeneous early-design artifacts. The conceptual contribution is a role-based agentic decomposition for performance model construction instantiated for PCM using GPT-5.4. AutoPCM extracts a structured knowledge base from the input documents and employs four specialized agents to populate all five PCM views without requiring a running system or manual modeling. We evaluated AutoPCM on TeaStore (9 use cases) and Corona-Warn-App (11 use cases). 205 of 220 generated PCM instances passed PCM metamodel validation and achieved a mean structural similarity of 0.86 (Jaccard coefficient) compared to manually authored reference models. Structural accuracy correlates with the clarity of the input artifacts. The primary deviation from reference models is over-generation of model elements that belong to the system but fall outside the scope of the particular use case, which is a defensible architectural decision that yields a richer performance model accommodating future extensions. As future work, we are planning an end-to-end evaluation that executes the generated models in PCM simulators (e.g., SimuLizar) and compares the results against those of the reference models.

Acknowledgements This work was funded by Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF), the German Research Foundation (DFG) –

SFB 1608 – 501798263, the Topic Engineering Secure Systems of the Helmholtz Association (HGF), and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the National Research Data Infrastructure – NFDI 52/1 – 501930651, and was supported by KASTEL Security Research Labs, Karlsruhe. We thank Fatih Çatalkaya for his work on PCM instance modeling and validation. This research was edited by our textician, Daniel Shea.

References

1. Adnan, B., Esposito, M., Taibi, D., Vaidhyanathan, K.: Can ai agents generate microservices? how far are we? arXiv preprint arXiv:2603.09004 (2026)
2. Altamimi, T., Zargari, M.H., Petriu, D.C.: Performance analysis roundtrip: automatic generation of performance models and results feedback using cross-model trace links. In: Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering. pp. 208–217 (2016)
3. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. Encyclopedia of software engineering pp. 528–532 (1994)
4. Basili, V.R., Weiss, D.M.: A methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering **10**(6), 728–738 (1984)
5. Brosig, F., Kounev, S., Krogmann, K.: Automated extraction of palladio component models from running enterprise java applications. In: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools. pp. 1–10 (2009)
6. Corona-Warn-App Team: Corona-warn-app, https://github.com/corona-warn-app
7. Cortellessa, V., Di Pompeo, D., Eramo, R., Tucci, M.: A model-driven approach for continuous performance engineering in microservice-based systems. Journal of Systems and Software **183**, 111084 (2022)
8. Eramo, R., Tucci, M., Di Pompeo, D., Cortellessa, V., Di Marco, A., Taibi, D.: Architectural support for software performance in continuous software engineering: A systematic mapping study. Journal of Systems and Software **207**, 111833 (2023)
9. Esposito, M., Li, X., Moreschini, S., Ahmad, N., Cerny, T., Vaidhyanathan, K., Lenarduzzi, V., Taibi, D.: Generative ai for software architecture. applications, challenges, and future directions. Journal of Systems and Software (2026)
10. Hummel, M., Fuchß, D., Corallo, S., Hagel, N., Keim, J., Koziolek, H., Reussner, R.: Replication Package: AutoPCM: Automated Generation of Architectural Performance Models from Early Design Artifacts using LLM Agents (2026). https://doi.org/10.5281/zenodo.20287035
11. Hummel, M., Fuchß, D., Corallo, S., Hagel, N., Kaushik, M., Keim, J., Reussner, R., Koziolek, H.: Garma: Generative architectural resource demand estimation for microservice applications. In: 2026 IEEE 23rd International Conference on Software Architecture Companion (ICSA-C) (2026), accepted for publication
12. Hummel, M., Hagel, N., Kaushik, M., Keim, J., Burger, E., Koziolek, H.: Llm-assisted microservice performance modeling. In: 16th Symposium on Software Performance (2025)
13. Jaccard, P.: Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. Bull Soc Vaudoise Sci Nat **37**, 241–272 (1901)
14. Kappler, T., Koziolek, H., Krogmann, K., Reussner, R.: Towards automatic construction of reusable prediction models for component-based performance engineering. In: Software Engineering 2008. Gesellschaft für Informatik e. V. (2008)

15. von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., Kounev, S.: TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In: Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems. pp. 223–236. IEEE (2018)
16. Koziolek, H., Happe, J.: A qos driven development process model for component-based software systems. In: International Symposium on Component-Based Software Engineering. pp. 336–343. Springer (2006)
17. Krogmann, K., Kuperberg, M., Reussner, R.: Using genetic search for reverse engineering of parametric behavior models for performance prediction. IEEE Transactions on Software Engineering **36**(6), 865–877 (2010)
18. Langhammer, M., Shahbazian, A., Medvidovic, N., Reussner, R.H.: Automated extraction of rich software models from limited system information. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA) (2016)
19. Li, C., Altamimi, T., Zargari, M.H., Casale, G., Petriu, D.: Tulsa: a tool for transforming uml to layered queueing networks for performance analysis of data intensive applications. In: International Conference on Quantitative Evaluation of Systems. pp. 295–299. Springer (2017)
20. Martens, A., Becker, S., Koziolek, H., Reussner, R.: An empirical investigation of the effort of creating reusable, component-based models for performance prediction. In: International Symposium on Component-Based Software Engineering (2008)
21. Mazkatli, M., Monschein, D., Armbruster, M., Heinrich, R., Koziolek, A.: Continuous integration of architectural performance models with parametric dependencies–the cipm approach. Automated Software Engineering **32**(2), 54 (2025)
22. Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A.: Modeling and simulating software architectures: The Palladio approach. MIT Press (2016)
23. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering **14**(2), 131–164 (2009)
24. Sallou, J., Durieux, T., Panichella, A.: Breaking the silence: the threats of using llms in software engineering. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: NIER (2024)
25. Smith, C.U.: Origins of software performance engineering: Highlights and outstanding problems. In: International Workshop on Software and Performances. pp. 96–118. Springer (2000)
26. Smith, C.U., Williams, L.G.: Performance solutions: a practical guide to creating responsive, scalable software, vol. 1. Addison-Wesley Reading (2002)
27. Tagliaferro, A., Corbo, S., Guindani, B.: Leveraging llms to automate software architecture design from informal specifications. In: 2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C). pp. 291–299 (2025)
28. Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Future of Software Engineering (FOSE '07). pp. 171–187. IEEE (2007)
29. Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (puma). In: Proceedings of the 5th international workshop on Software and performance. pp. 1–12 (2005)
30. Yang, Y., Chen, B., Chen, K., Mussbacher, G., Varró, D.: Multi-step iterative automated domain modeling with large language models. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems. pp. 587–595 (2024)
31. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: ReAct: Synergizing reasoning and acting in language models. In: International Conference on Learning Representations (ICLR) (2023)