AUTONOMOUS MEDICAL IMAGING PIPELINE SYNTHESIS WITH VALIDATION-GATED REFINEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models enable increasingly autonomous machine learning workflows, but reliable end-to-end pipeline construction remains difficult in specialised domains such as medical imaging. Pipeline synthesis requires coordinated decisions over dataset interpretation, preprocessing, model design, training, and integration, while an incorrect intermediate decision can invalidate subsequent stages. We introduce MedPilot, a validation-gated agentic framework for autonomous medical imaging pipeline synthesis. Rather than generating a pipeline in a single pass, MedPilot decomposes synthesis into specialised stages and validates intermediate artefacts before they are propagated downstream. Failed candidates receive structured feedback and are iteratively repaired, while validated solutions are accumulated in an adaptive knowledge base and reused as references for related tasks. We evaluate MedPilot on six medical imaging benchmarks spanning supervised and semi-supervised segmentation and 3D classification. MedPilot achieves zero human intervention across all evaluated runs, with Dice scores of up to 0.945 on segmentation and 98.6% accuracy on classification, while consistently outperforming the evaluated baselines. These results demonstrate the effectiveness of validation-gated refinement for reliable and autonomous medical imaging pipeline synthesis.

1 INTRODUCTION

The rapid progress of large language models (LLMs) has created new opportunities for automating increasingly complex computational workflows (Jin et al., 2024; Jiang et al., 2024; He et al., 2025). Trained on large-scale code corpora, modern LLMs achieve high performance on function synthesis and repository-level code repair (Chen et al., 2021; Austin et al., 2021; Jimenez et al., 2024). However, these results do not yet demonstrate that LLM-based systems can reliably construct end-to-end machine learning pipelines from task requirements and dataset artefacts, where multiple interdependent design decisions must be coordinated under domain-specific constraints.

This gap is particularly sharp for AutoML in specialised domains. Medical imaging is a representative example: an end-to-end pipeline must jointly handle dataset analysis, modality-specific preprocessing, the choice between 2D and 3D designs, training and inference configuration, and specialised formats such as NIfTI (Li et al., 2016) and DICOM (Mildenberger et al., 2002). Small errors in any of these decisions can substantially degrade downstream performance, making medical imaging a stringent testbed for whether agentic systems can synthesise end-to-end pipelines under domain constraints rather than solve isolated coding tasks.

Prior automation methods address parts of this problem but leave the full task-to-pipeline synthesis loop unresolved. In medical imaging, NAS methods (Liu et al., 2021; Ren et al., 2021; Wang et al., 2024) automate model design within predefined search spaces. nnU-Net (Isensee et al., 2021) goes further by extracting dataset fingerprints and automatically configuring preprocessing, network architecture, training, and inference through a predefined self-configuring framework. Foundation models such as SAM and MedSAM (Kirillov et al., 2023; Ravi et al., 2024; Ma et al., 2024a) reduce task-specific development through prompt-driven segmentation inference. These methods provide strong automation within predefined model families, configuration rules, or inference interfaces, but do not perform open-ended synthesis of executable pipeline implementations with iterative, stage-specific validation and repair.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

LLM-based agentic systems broaden the scope of automation but remain limited for specialised pipeline synthesis. Single-LLM (OpenAI, 2026; Anthropic, 2026a; Google Deepmind, 2026; Qwen-Team, 2026; DeepSeek-AI, 2026) and IDE-based coding tools (Github, 2025; Google, 2025; CursorTeam, 2024) require detailed manual instructions to convey task-specific background (Li et al., 2024) and let hallucinated intermediate decisions propagate without validation (Shin et al., 2025). General multi-agent coding frameworks such as CAMEL (Li et al., 2023), MetaGPT (Hong et al., 2023), and MaCTG (Zhao et al., 2026) decompose project development into collaborative agent roles, thereby improving task organisation compared to single-agent generation. However, their coordination is primarily oriented toward generic coding workflows rather than domain-constrained medical imaging pipeline synthesis. Agentic AutoML systems such as DS-Agent (Guo et al., 2024) and AutoML-Agent (Trirat et al., 2024) formalise multi-step workflow generation across planning, execution, and feedback, but remain general-domain frameworks that rely primarily on execution feedback or manually provided scaffolds. Medical-domain agentic systems are closer in application: M^{3}Builder (Feng et al., 2025) targets medical ML workflow construction but relies on predefined datasets and template-supported pipelines, while VoxelPrompt (Hoopes et al., 2024) focuses on inference-time radiological task execution rather than automated pipeline development or code generation. No existing system achieves fully automated medical imaging pipeline synthesis while maintaining strong downstream performance.

In this work, we propose **MedPilot**, an agentic AutoML framework that builds upon Claude-Opus-4.7 (Anthropic, 2026b), which approaches end-to-end pipeline synthesis as a constrained search problem. A pipeline's validity predicate is factorised into hierarchical sub-predicates spanning the planning and coding phases, turning validation into a sequence of feasibility-shrinking operators that admit only validated artefacts into shared memory. A multi-agent role decomposition separates generation from validation, isolates per-agent domain knowledge and context, and parallelises module synthesis; an adaptive knowledge base containing previously validated solutions accumulates experience and warm-starts refinement for related tasks. Unlike fixed pipeline templates, retrieved solutions are used only as adaptable references: agents may modify their structure and implementation according to the new task and dataset, and all adapted artefacts must pass the same validation process as pipelines synthesised from scratch. Across six medical imaging benchmarks, MedPilot achieves full automation with Dice scores of up to 0.945 on segmentation and 98.6% accuracy on classification.

The main contributions of this work are as follows:
1) We present MedPilot, an agentic AutoML framework for **fully automated end-to-end pipeline synthesis**. Across six medical imaging benchmarks, MedPilot requires no human interaction while delivering strong downstream performance across all datasets.
2) We propose a **validation-gated agentic refinement** formulation that casts pipeline synthesis as a constrained search problem and factorises the validity predicate into hierarchical sub-predicates acting as feasibility-shrinking operators. The factorisation supports incremental enforcement of correctness and the local diagnosis of failures.
3) We introduce a **cross-project knowledge accumulation** mechanism that treats the cumulative store of validated pipelines as a non-parametric prior over the pipeline space. Retrieval against this prior warm-starts refinement for related tasks, while re-validation guards against transferring incompatible prior solutions.

2 METHODOLOGY

We formulate end-to-end pipeline synthesis as a *constrained search problem* and approximate its solution through *agentic iterative refinement*. As illustrated in Figure 1, given a task specification and dataset, MedPilot proceeds through two coupled synthesis stages. In the planning stage, the Dataset Analyser extracts data-level descriptors, and the Architect uses these descriptors together with domain knowledge to propose an abstract pipeline design. The design is validated before entering the coding stage, where the Coding Router decomposes it into implementation subtasks, assigns them to specialised Coders, and assembles the resulting modules into a candidate executable pipeline. Throughout this process, failed validation produces diagnostic feedback that is appended to the relevant agent context, while validated artefacts are stored in shared memory for downstream agents. We formalise this process as follows. Section 2.1 formalises the constrained synthesis objective. Section 2.2 introduces the agent population, structured context, and validation-driven update rule.

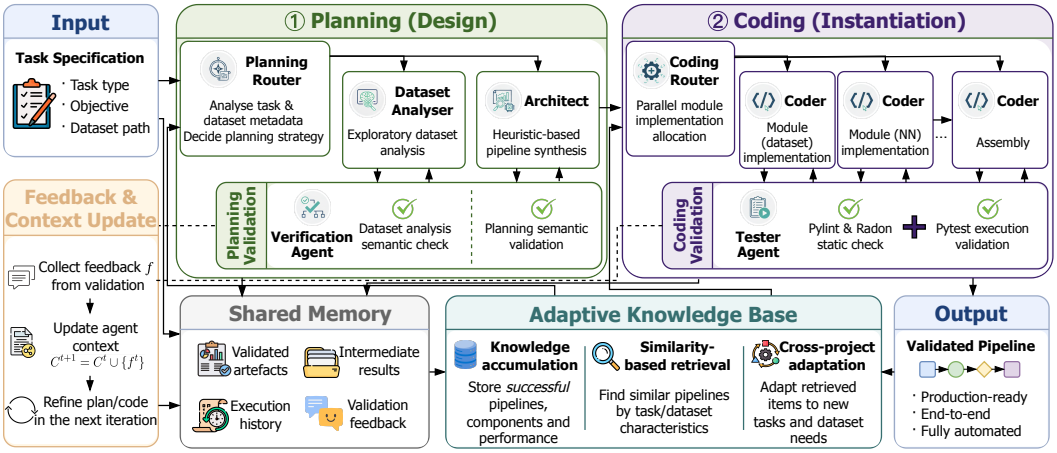

Figure 1: Overview of MedPilot's framework. A Planning Router coordinates the Dataset Analyser and Architect to extract dataset characteristics and produce an abstract pipeline design. A Coding Router assigns implementation subtasks to specialised Coders, whose outputs are assembled into an executable pipeline. Validators are embedded across planning, coding, and assembly stages to decide which artefacts are committed to propagate, while the shared memory stores validated intermediate results among agents and the adaptive knowledge base accumulates and retrieves validated prior solutions for related tasks.

Section 2.3 factorises the validity predicate into hierarchical sub-predicates. Section 2.4 formalises cross-project transfer as a non-parametric prior over the pipeline space.

2.1 Constrained Pipeline Synthesis

Let D denote a dataset and T a task specification. We define $\mathcal{P}$ as the space of admissible end-to-end pipelines, where each $P \in \mathcal{P}$ is a directed composition of stages including data ingestion, preprocessing, model architecture, training, and inference, and induces an executable program $\mathrm{Exec}(P; D)$. Structural, executional, and domain constraints are aggregated into a validity predicate $\Phi(P, D, T) \in \{\mathrm{true}, \mathrm{false}\}$. We formulate pipeline synthesis as the problem of finding a feasible pipeline:

$$P^\star \in \mathcal{F}(D, T) = \{P \in \mathcal{P} \mid \Phi(P, D, T) = \mathrm{true}\}\,. \tag{1}$$

Finding a feasible P is challenging because $\mathcal{P}$ is combinatorial and some validity conditions, particularly executional ones, can only be evaluated after generating and partially executing candidate artefacts. MedPilot therefore constructs $P^\star$ through progressive constraint enforcement. The validity predicate Φ is decomposed into hierarchical sub-predicates that are evaluated incrementally during synthesis, while failed checks provide diagnostic feedback for local repair.

2.2 Agentic Iterative Refinement

We model pipeline synthesis as a sequence of partial decisions made by a population of specialised agents $\{A_i\}_{i=1}^{N}$, each induced by a conditional generative model $g_{A_i}(\cdot \mid C_i)$ over a sub-region of $\mathcal{P}$. Specialisation is achieved through differing contexts rather than differing parameters; in our instantiation each agent is a prompted LLM with a role-specific context.

Structured context. The context C_i is the only object on which g_{A_i} conditions, and we structure it as

$$C_i \;=\; \langle\, P_i,\; \mathcal{M},\; K_i,\; K_{\text{ada}},\; H_i,\; V_i \,\rangle, \tag{2}$$

where P_i is the role prompt, $\mathcal{M}$ is shared memory containing validated artefacts, K_i is agent-specific domain knowledge, K_{ada} is the cross-project adaptive knowledge base (Section 2.4), H_i is the agent's interaction trace, and V_i accumulates validation feedback. The factorisation separates static priors $(P_i, K_i, K_{\text{ada}})$ from dynamic state $(\mathcal{M}, H_i, V_i)$, a separation we exploit later for warm-starting via retrieval.

Validation-driven update. At iteration t, the active agent emits a candidate $o_i^{(t)} = g_{A_i}(\cdot \mid C_i^{(t)})$, which is evaluated by a validator $\mathcal{V}$ that returns either `pass` or a pair $(\texttt{fail}, f_i^{(t)})$ where $f_i^{(t)}$ is a structured diagnostic localising the failure. The update operator $\mathcal{U}$ inherits the static components $(P_i, K_i, K_{\text{ada}})$, extends H_i with the current invocation, and updates the validated state $(\mathcal{M}, V_i)$ according to

$$\big(\mathcal{M}^{(t+1)},\ V_i^{(t+1)}\big) \;=\; \begin{cases} \big(\mathcal{M}^{(t)} \cup \{o_i^{(t)}\},\ V_i^{(t)}\big) & \text{if } \mathcal{V}(o_i^{(t)}) = \texttt{pass}, \\ \big(\mathcal{M}^{(t)},\ V_i^{(t)} \cup \{f_i^{(t)}\}\big) & \text{otherwise.} \end{cases} \tag{3}$$

Each validation stage allows at most three repair attempts ($R_{\max} = 3$). Failed candidates are regenerated using the validator feedback; if all retries fail, the validation check is bypassed and synthesis continues. In our experiments, all validation failures were resolved within one retry, so this fallback was never triggered.

Two properties follow. First, under normal refinement, M admits artefacts only after validation, so downstream agents condition on artefacts that have passed their corresponding stage-specific checks. The bounded retry fallback is used only to prevent indefinite execution when a validation failure cannot be resolved. Second, failed outputs are not committed; instead, the diagnostic $f_i^{(t)}$ is appended to V_i, which sharpens the next conditional $g_{A_i}(\cdot \mid C_i^{(t+1)})$ toward feasible regions of $\mathcal{P}$. Refinement is therefore monotone in the validated state and locally corrective in the unvalidated state.

Agent role decomposition. The agent population $\{A_i\}_{i=1}^N$ is a structural choice with three theoretical consequences. First, distributing generator and validator roles across distinct agents separates generation from evaluation, allowing V to act as an external check rather than relying on self-evaluation by the generating agent. Second, each agent's conditional $g_{A_i}(\cdot \mid C_i)$ is restricted by a focused context with role-specific knowledge K_i, narrowing the generation distribution relative to a single agent that must span multiple roles. Third, the predicate factorisation of Section 2.3 allows module-level validations $\phi_{\text{stat}}(c_i) \wedge \phi_{\text{exec}}(c_i)$ to be performed separately when their component dependencies permit, exposing parallelism across coder agents.

2.3 Hierarchical constraint factorisation

The validity predicate admits a natural factorisation along the temporal order of pipeline construction:

$$\Phi(P, \mathcal{D}, \mathcal{T}) \;=\; \Phi_{\text{plan}}(P_{\text{arch}}, \mathcal{D}, \mathcal{T}) \;\wedge\; \Phi_{\text{code}}(P, P_{\text{arch}}), \tag{4}$$

where P_{arch} is the abstract pipeline design (modules, interfaces, hyperparameters, dataflow) and P is its concrete realisation as executable code. Let $\hat{\mathbf{d}}$ denote the dataset descriptors generated by the Dataset Analyser from $\mathcal{D}$. Each factor decomposes further into successively stricter sub-predicates:

$$\Phi_{\text{plan}} \;=\; \phi_{\text{data}}(\hat{\mathbf{d}}, \mathcal{D}) \;\wedge\; \phi_{\text{ctx}}(P_{\text{arch}}, \hat{\mathbf{d}}, \mathcal{M}, K_{\text{arch}}) \;\wedge\; \phi_{\text{form}}(P_{\text{arch}}), \tag{5}$$

$$\Phi_{\text{code}} \;=\; \bigwedge_i \big[\, \phi_{\text{stat}}(c_i) \;\wedge\; \phi_{\text{exec}}(c_i) \,\big] \;\wedge\; \phi_{\text{asm}}(\text{Compose}(\{c_i\}))\,. \tag{6}$$

ϕ_{data} checks whether the generated descriptors $\hat{\mathbf{d}}$ are consistent with the underlying dataset $\mathcal{D}$; ϕ_{ctx} checks whether P_{arch} is compatible with these validated descriptors and with domain conventions encoded in K_{arch}, where K_{arch} denotes the Architect's role-specific domain knowledge; ϕ_{form} verifies process-level well-formedness of P_{arch}, including deadlock freedom and required reachability or ordering properties. A fixed rule-based translator maps pipeline components to CSP# processes, inter-component dependencies to guarded transitions and state variables, and workflow requirements to model-checking assertions, which are then verified using Process Analysis Toolkit. On the coding side, ϕ_{stat} enforces syntactic correctness, dependency consistency, and quality thresholds on each module c_i; ϕ_{exec} enforces functional correctness on agent-generated unit tests; and ϕ_{asm} enforces interface and dataflow consistency at composition time, capturing failure modes invisible at the per-module level (e.g., a producer emitting tensors under key k while its consumer expects k').

Why factorise. Two properties make this factorisation more than a notational convenience. *(i) Monotone feasibility shrinkage.* Each additional conjunct in Eqs. 5– 6 can only reduce or preserve the feasible candidate set; for example, $\{P_{\text{arch}} : \phi_{\text{data}} \wedge \phi_{\text{ctx}}\} \subseteq \{P_{\text{arch}} : \phi_{\text{data}}\}$. Thus, validation progressively constrains the set of candidates admitted to later synthesis stages. Refinement therefore

operates on progressively constrained candidate sets, which can avoid unnecessary downstream regeneration by identifying invalid intermediate artefacts before full pipeline assembly. *(ii) Local diagnosability.* The verdict of each ϕ carries a feedback term that identifies the failing sub-clause, which lets the update operator $\mathcal{U}$ in Eq. 3 localise corrections to the responsible agent rather than restarting the whole synthesis. Locality is what distinguishes hierarchical validation from the single execution check used by prior agentic AutoML systems and is the mechanism by which Φ is enforced incrementally rather than only retrospectively.

2.4 Cross-project adaptive synthesis

End-to-end synthesis from scratch is wasteful when a closely related task has already been solved. We capture cross-project transfer as a non-parametric prior over $\mathcal{P}$ instantiated by an adaptive knowledge base

$$K_{\text{ada}} \;=\; \left\{\, (P^{(j)},\, \xi^{(j)},\, \rho^{(j)}) \,\right\}_{j=1}^{|K_{\text{ada}}|}, \tag{7}$$

where $\xi^{(j)}$ is a structured task descriptor (modality, task type, dimensionality, supervision regime, dataset statistics) and $\rho^{(j)}$ denotes the downstream performance achieved by pipeline $P^{(j)}$. The knowledge base is grown only by validated solutions:

$$K_{\text{ada}} \leftarrow K_{\text{ada}} \cup \{(P, \xi, \rho)\} \quad \text{iff} \quad \Phi(P, D, T) = \text{true}, \tag{8}$$

The knowledge base is populated with pipelines that complete the synthesis process and pass the corresponding validation stages, together with their task descriptors and downstream performance.

Retrieval as warm start. For a new task with descriptor ξ_{new}, we define the retrieval set

$$\mathcal{N}(\xi_{\text{new}}) = \mathcal{R}_{\text{sem}}\left(\xi_{\text{new}};\, K_{\text{ada}}\right), \tag{9}$$

where $\mathcal{R}_{\text{sem}}$ is a semantic retrieval operator that ranks the cases in K_{ada} according to their relevance to the structured descriptor ξ_{new} and returns a subset $\mathcal{N}(\xi_{\text{new}})$ of relevant prior solutions. Retrieved cases serve only as references for task-specific adaptation, and all adapted artefacts re-enter the same validation-driven refinement process.

Closure under validation. Crucially, retrieved artefacts are never admitted to $\mathcal{M}$ or K_{ada} without re-entering the refinement loop of Section 2.2: every adapted candidate must satisfy Φ before it is committed. This invariant guards against invalid or incompatible transfer: a retrieved solution that violates the constraints of ξ_{new} either fails validation and is rejected or is corrected through the same diagnostic feedback used for from-scratch synthesis. The result is a case-based meta-learning scheme over a structured solution space, in which the case base is grown only by validated experience and similarity-driven retrieval supplies an inductive bias that is empirically expensive to encode in either prompts or frozen weights.

3 Experiments

We evaluate MedPilot on six medical imaging benchmarks against autonomous workflow and agentic AutoML baselines. Section 3.1 describes the experimental setup, Section 3.2 presents the main performance and autonomy results, and Section 3.3 provides ablations, medical-automation positioning, and cost analysis.

3.1 Setup

Datasets. We incorporate six widely used medical imaging benchmarks covering segmentation (2D and 3D, supervised and semi-supervised) and classification across CT and MRI, spanning a difficulty gradient from low-resolution 3D classification to high-cardinality semi-supervised volumetric segmentation as shown in Table 1.

Baselines and MedPilot configurations. We compare MedPilot against representative systems capable of autonomously constructing machine learning workflows. The baselines span three categories: general multi-agent coding frameworks, including MetaGPT X (MGX) (FoundationAgents, 2025) and MaCTG (Zhao et al., 2026); agentic AutoML frameworks, including AutoML-Agent (Trirat et al., 2024), M^{3}Builder (Feng et al., 2025), and MLZero (Fang et al., 2026); and IDE-based coding assistants, including GitHub Copilot (Github, 2025), Cursor (CursorTeam, 2024), and Antigravity (Google,

Table 1: Details on the benchmarking datasets.

Dataset	Task	Mod.	#Samples	#Classes	Avg. dim.	Salient challenge
FLARE22 (Ma et al., 2024b)	Semi-sup. 3D Seg.	CT	2,050	14	$512\times512\times196$	Large unlabelled pool
ACDC (Bernard et al., 2018)	Sup. 3D Seg.	MRI	300	4	$215\times228\times10$	Anisotropic spacing
BTCV (Landman et al., 2015)	Sup. 3D Seg.	CT	50	14	$512\times512\times123$	Few cases, class imbalance
FLARE21 (Ma et al., 2022)	Sup. 3D Seg.	CT	361	5	$512\times512\times126$	Mid-difficulty reference
Promise12MSBench (Kuş & Aydin, 2024)	Sup. 2D Seg.	MRI	1,473	2	512×512	2D, binary
OrganMNIST3D (Yang et al., 2023)	Sup. 3D Cls.	CT	1,742	11	$64\times64\times64$	Low-resolution volumetric

2025). To ensure a consistent comparison, all methods receive the same task specification covering dataset analysis, pipeline design, implementation, and integration, together with access to the command line and file system, permission to install required dependencies, and the ability to iteratively revise their solutions based on execution feedback.

Metrics. We evaluate each method along two axes: downstream performance and autonomy. For segmentation tasks, downstream performance is measured by the Dice coefficient (Zijdenbos et al., 1994). For class c, Dice is computed as

$$\mathrm{Dice}_c = \frac{2\sum_v \mathbb{1}[\hat{y}_v = c]\mathbb{1}[y_v = c]}{\sum_v \mathbb{1}[\hat{y}_v = c] + \sum_v \mathbb{1}[y_v = c] + \epsilon}, \tag{10}$$

where v indexes pixels or voxels, $\hat{y}$ is the predicted label map, y is the ground truth, and ϵ is a small constant for numerical stability. We report mean Dice averaged over foreground classes and test cases. For classification, we report accuracy:

$$\mathrm{Acc} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}[\hat{y}_i = y_i], \tag{11}$$

where N is the number of test samples.

To quantify autonomy, we define *Intervention Burden* (IB) as

$$\mathrm{IB}(r) = \sum_{e\in\mathcal{E}_r} \mathbb{1}[\mathrm{human}(e) \wedge \mathrm{type}(e) \in \{\mathrm{E1}, \mathrm{E2}, \mathrm{E3}\}]\,. \tag{12}$$

where $\mathcal{E}_r$ is the set of synthesis-time events in run r, $\mathrm{human}(e)$ indicates whether event e involves human intervention, and $\mathrm{type}(e)$ assigns the intervention to one of the categories $\{\mathrm{E1}, \mathrm{E2}, \mathrm{E3}\}$. Specifically, E1 covers direct prompt interactions, including additional instructions, relaying execution errors, or providing corrections; E2 covers supplying additional templates or skeleton code; and E3 covers manual modification of generated scripts or configuration files. Each discrete human corrective action is counted as one intervention, while errors resolved autonomously through the method's own execution and retry process are not counted. A run is fully autonomous when $\mathrm{IB} = 0$.

Each (method, dataset) pair is evaluated over three independent trials. We report the mean and standard deviation across the three trials for both downstream performance and intervention burden, capturing both the average effectiveness of each method and its run-to-run variability.

3.2 Experimental Results

We present the main results in two parts, first examining the quality of the synthesised pipelines and then their autonomy during synthesis.

3.2.1 Pipeline Quality

Downstream performance. We first examine the downstream model quality achieved by the synthesised pipelines on each benchmark. Figure 2 reports the mean performance and standard deviation over three independent trials. MedPilot achieves the highest mean performance on all six benchmarks, demonstrating consistent effectiveness across both segmentation and classification tasks. In contrast, the strongest competing method varies substantially across datasets, suggesting that no individual baseline provides comparable robustness across the full benchmark suite.

On the five segmentation benchmarks, MedPilot consistently outperforms the strongest baseline. On FLARE22, MedPilot achieves a Dice of 0.8094 ± 0.0114, compared with 0.7598 ± 0.0509

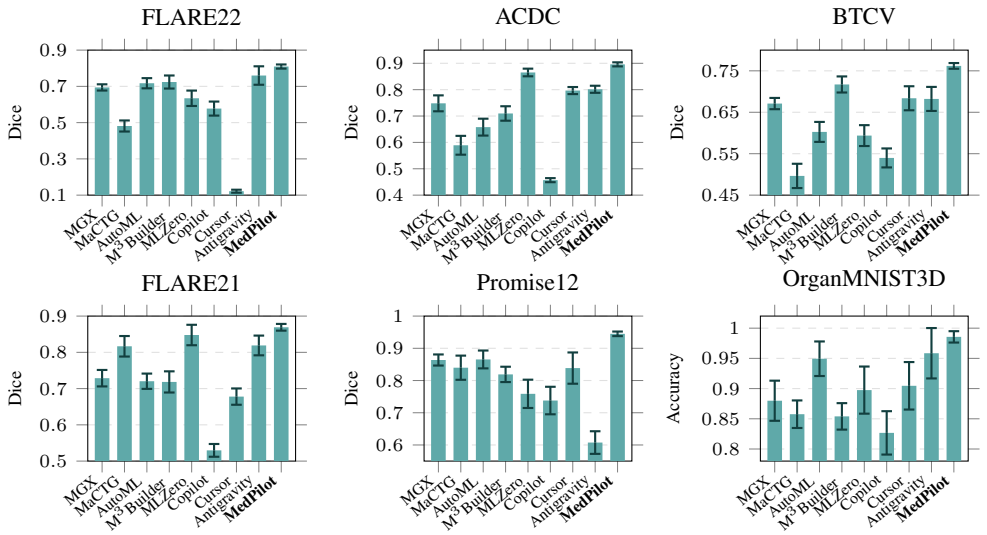

Figure 2: Downstream performance across six medical imaging benchmarks. Bars report mean performance over three independent trials, with error bars denoting standard deviation.

for Antigravity. On ACDC, it reaches 0.8960 ± 0.0077, exceeding MLZero (0.8652 ± 0.0145), while on BTCV it obtains 0.7619 ± 0.0069 compared with 0.7171 ± 0.0194 for M^{3}Builder. The advantage remains on FLARE21, where MedPilot achieves 0.8691 ± 0.0093 versus 0.8478 ± 0.0282 for MLZero, and is particularly pronounced on Promise12, where MedPilot reaches 0.9449 ± 0.0071 compared with 0.8654 ± 0.0277 for AutoML-Agent.

The same trend extends to classification. On OrganMNIST3D, MedPilot achieves an accuracy of 0.9856 ± 0.0094, outperforming the strongest baseline, Antigravity, at 0.9585 ± 0.0417. Across the benchmarks, MedPilot also exhibits relatively small run-to-run variation, indicating that its advantage is not driven by isolated high-performing trials. These results suggest that the validation-gated, expert-grounded synthesis process improves not only the quality of the resulting pipelines but also their consistency across heterogeneous medical imaging tasks.

Pipeline-design quality. Numerical performance alone does not reveal whether the synthesised pipelines follow appropriate medical-imaging design practice. Table 2 compares the BTCV pipelines across preprocessing, sampling, augmentation, and architecture choices. MedPilot produces the most complete volumetric pipeline, combining re-orientation, resampling, padding, intensity normalisation and perturbation, patch-based training, geometric augmentation, and a 3D architecture.

Table 2: BTCV pipeline-design profiles across autonomous frameworks and coding assistants.

Method	Spatial standardisation	Intensity handling	Sampling strategy	Geometric augmentation	Architecture	Overall profile
MGX	Re-orientation + resampling	Normalisation + intensity jitter	Patch-based	Rotation + flipping	UNet3D + RN	Broad but generic
MaCTG	Re-orientation + resampling	Normalisation + jitter/noise/ smoothing	Full-volume / implicit	Rotation + flipping	UNet3D	Heavy augmentation
AutoML-Agent	Re-orientation + resampling	Normalisation	Patch-based	None specified	UNet3D	Conservative implementation
M^{3}Builder	Padding	Normalisation + intensity jitter	Patch-based	Rotation + flipping	UNet3D (templated)	Template-like
MLZero	Resampling	Normalisation	Patch-based	None specified	UNet3D + RN	Basic volumetric
Copilot	None specified	Normalisation	None specified	None specified	UNet3D	Minimal pipeline
Cursor	Resize	Normalisation	None specified	Rotation + flipping + affine	UNet2D	2D-oriented pipeline
Antigravity	Re-orientation + resampling	Normalisation	Patch-based	None specified	UNet3D	Volumetric, limited augmentation
MedPilot	**Re-orientation + resampling + padding**	**Normalisation + jitter/noise**	**Patch-based**	**Rotation + flipping**	**UNet3D**	**Complete volumetric**

RN = ResNet. "Re-orientation" denotes orientation normalisation.

In contrast, the baselines typically omit or simplify at least one important component. MGX and MaCTG include several domain-relevant operations but remain less complete overall; AutoML-Agent omits geometric augmentation; M^{3}Builder uses limited spatial standardisation; and MLZero relies on fixed-shape resampling without explicit orientation or spacing normalisation. Among IDE-based assistants, Copilot produces a minimal pipeline, Cursor adopts a 2D U-Net, and Antigravity lacks explicit augmentation. Overall, the results suggest that MedPilot's advantage comes from composing a more complete set of domain-appropriate design choices rather than from any single component.

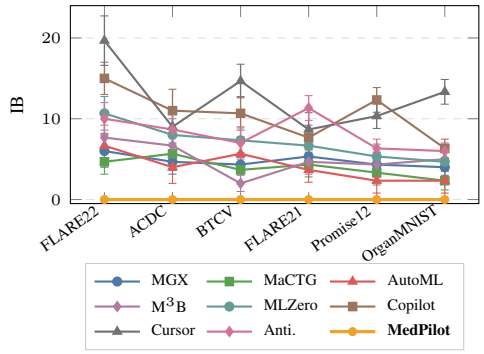

Figure 3: Dataset-wise Intervention Burden (mean $\pm$ std.; lower is better).

3.2.2 Autonomy Analysis

Figure 3 reports dataset-wise Intervention Burden (IB) over three independent trials. MedPilot maintains IB $= 0$ on all six benchmarks, completing the full pipeline-synthesis process without human corrective intervention. In contrast, every baseline requires intervention, and the burden varies substantially across both methods and datasets. Agentic frameworks generally require fewer corrections than IDE-based coding assistants; for example, M^{3}Builder ranges from 2.00 ± 1.00 interventions on BTCV to 7.67 ± 1.53 on FLARE22, while MLZero reaches 10.67 ± 2.08 on FLARE22.

The gap is larger for coding assistants, with Copilot and Cursor requiring up to 15.00 ± 2.00 and 19.67 ± 3.06 interventions, respectively. These results show that generating executable code alone is insufficient for fully autonomous pipeline construction, as intermediate inconsistencies still frequently require human recovery.

MedPilot's zero-intervention performance indicates that its validation-gated execution can detect and resolve such failures internally before they propagate downstream.

3.3 Additional Analysis

We conduct additional analyses to examine why MedPilot improves autonomy and performance, how it relates to existing medical imaging automation methods, and whether its automation cost is practical relative to manual pipeline development.

Ablation study. We ablate steps in validation predicates, context components, and agent decomposition to quantify their contributions to autonomy and performance. Figure 4 reports the increase in Intervention Burden, ΔIB, and the decrease in downstream performance, ΔPerf., relative to full MedPilot.

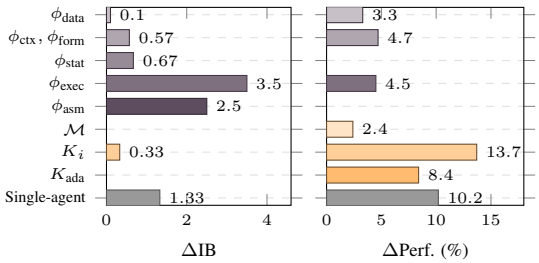

Figure 4: Per-component ablation of MedPilot.

Validation predicates affect both autonomy and downstream performance, although their effects differ by stage. Removing execution validation ϕ_{exec} and assembly validation ϕ_{asm} produces the largest increases in IB (+3.50 and +2.50, respectively), indicating that runtime and inter-module failures are major sources of human-visible intervention. In contrast, removing planning-level validation, particularly $\phi_{\text{ctx}}/\phi_{\text{form}}$ and ϕ_{data}, causes noticeable performance degradation, showing that early validation also influences the quality of the synthesised pipeline. Context components primarily affect downstream performance: removing shared memory M, adaptive knowledge K_{ada}, and agent-specific domain knowledge K_i reduces performance by 2.4, 8.4, and 13.7 percentage points, respectively. The reported ΔPerf. for K_{ada} is averaged across the benchmark sequence; its effect is therefore concentrated on later tasks that can retrieve prior cases, while the first task is evaluated with an initially empty adaptive knowledge base. Collapsing MedPilot into a

single-agent configuration affects both autonomy and performance (+1.33 IB and a 10.2-percentage-point decrease in performance), supporting the contribution of role specialisation and stage-specific context.

Comparison with medical imaging automation methods. Figure 5 positions MedPilot relative to representative medical imaging automation and agentic AutoML methods. Existing medical imaging automation methods provide strong automation within narrower workflow regions: NAS focuses on architecture search, nnU-Net self-configures segmentation pipelines from dataset fingerprints within a fixed framework, and MedSAM supports prompt-driven segmentation inference. Agentic AutoML methods, including M^{3}Builder, MLZero, and AutoML-Agent, broaden the automation scope toward workflow- or pipeline-level construction, but their adaptation remains constrained by general-domain assumptions, predefined templates, or limited task-specific validation.

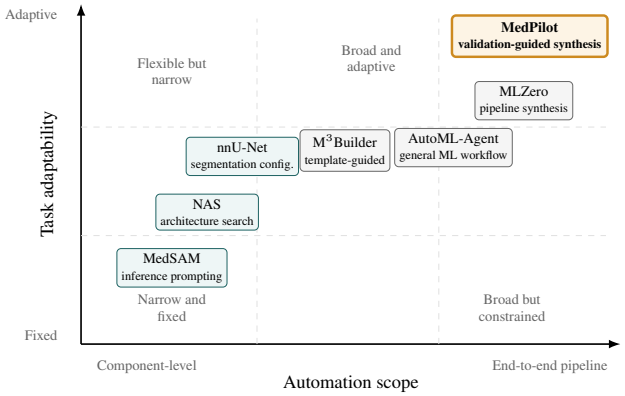

Figure 5: Qualitative comparison with representative medical imaging automation and agentic AutoML methods.

MedPilot targets a different point in this design space. It extends automation toward end-to-end medical imaging pipeline synthesis while retaining task adaptability through validation-guided refinement and reuse of validated prior solutions. This distinction is important: MedPilot is not simply another segmentation configuration method or generic workflow agent, but a framework for synthesising executable medical imaging pipelines from task and dataset inputs.

Cost-efficiency analysis. Table 3 provides an indicative comparison between MedPilot and published high-performing reference solutions rather than a controlled performance baseline. The reference scores are drawn from representative challenge entries or prior studies and may differ in training data, evaluation splits, and optimisation protocols. MedPilot remains below the published reference on FLARE22 and BTCV, is close on FLARE21, and numerically exceeds the reported references on Promise12 and OrganMNIST3D.

Table 3: Indicative comparison with published reference solutions and estimated expert development effort.

Dataset	Published reference	MedPilot
FLARE22	0.906 (Huang et al., 2022)	0.809
ACDC	0.936 (Zhao et al., 2022)	0.896
BTCV	0.921 (Valanarasu et al., 2023)	0.762
FLARE21	0.895 (Zhang et al., 2021)	0.869
Promise12	0.911 (Kuş & Aydin, 2024)	0.945
OrganMNIST3D	0.965 (Jain et al., 2024)	0.986
Dev. time	16–40 h†	30–40 min
Dev. cost	\$1,600–\$6,000†	\$20–\$40

†Engineering estimate based on the authors' practical experience.

The development-effort figures are engineering estimates based on the authors' practical experience in medical-imaging algorithm development. MedPilot synthesises a complete pipeline in approximately 30–40 minutes at an API cost of \$20–\$40, compared with an estimated 16–40 hours and \$1,600–\$6,000 of expert development effort. These figures illustrate the trade-off between expert-level optimisation and low-cost automated pipeline construction rather than a controlled comparison with human-designed solutions.

4 Conclusion

In this work, we presented MedPilot, an agentic AutoML framework for fully automated end-to-end medical imaging pipeline synthesis. MedPilot combines role-specialised agents, structured context management, hierarchical validation, and adaptive reuse of validated prior pipelines. Across six benchmarks, MedPilot achieves zero Intervention Burden while maintaining strong downstream performance against autonomous workflow and agentic AutoML baselines. Additional analyses examine the contribution of MedPilot's individual components, its positioning relative to existing medical imaging automation methods, and its practical synthesis cost relative to manual expert development.

AI USE STATEMENT

In this work, we used generative AI tools for one research task requiring disclosure: assisting with parts of the implementation of the research artefacts. We also used generative AI tools for recommended-disclosure tasks related to manuscript preparation, specifically to improve the clarity, readability, grammar, and wording of the paper.

We did not use generative AI tools for other research tasks requiring disclosure, including the development of theoretical models or mathematical claims, proof writing, hypothesis generation, methodology or experiment design, data cleaning or analysis, or interpretation of results. All AI-assisted code and text were reviewed by the authors. In particular, AI-assisted artefact code was manually inspected, tested, and integrated by the authors, and AI-assisted writing edits were checked to ensure that they accurately reflected the intended technical content and did not introduce unsupported claims or alter the reported results. We take full responsibility for the final content of this work, including all text, claims, code, and artefacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The paper provides an anonymised code repository with MedPilot implementation and reproduction instruction: `https://anonymous.4open.science/r/MedPilot-551C/`. The evaluated datasets are publicly accessible. Appendices B and C provide the experimental environment, baseline prompting, and evaluation protocol needed to reproduce the main results. The exact prompts and agent definitions are in the code repository.

REFERENCES

Anthropic. Introducing agent skills, October 2025. URL `https://claude.com/blog/skills`.

Anthropic. Introducing claude opus 5, June 2026a. URL `https://www.anthropic.com/news/claude-opus-5`.

Anthropic. Introducing claude opus 4.7, April 2026b. URL `https://www.anthropic.com/news/claude-opus-4-7`.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Olivier Bernard, Alain Lalande, Clement Zotti, Frederick Cervenansky, Xin Yang, Pheng-Ann Heng, Irem Cetin, Karim Lekadir, Oscar Camara, Miguel Angel Gonzalez Ballester, et al. Deep learning techniques for automatic mri cardiac multi-structures segmentation and diagnosis: is the problem solved? *IEEE transactions on medical imaging*, 37(11):2514–2525, 2018.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

CursorTeam. Cursor docs, 2024. URL `https://cursor.com/docs`.

DeepSeek-AI. Deepseek-v4.1-flash: Pushing the limits of kv cache compression, September 2026. URL `https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash/blob/main/DeepSeek_V41_Tech_Report.pdf`.

eiondb. Shared memory storage for multi-agent systems, 2025. URL `https://github.com/eiondb/eion`.

Haoyang Fang, Boran Han, Nick Erickson, Xiyuan Zhang, Su Zhou, Anirudh Dagar, Jiani Zhang, Ali Caner Turkmen, Tony Hu, Huzefa Rangwala, et al. Mlzero: A multi-agent system for end-to-end machine learning automation. *Advances in Neural Information Processing Systems*, 38: 69001–69070, 2026.

Jinghao Feng, Qiaoyu Zheng, Chaoyi Wu, Ziheng Zhao, Ya Zhang, Yanfeng Wang, and Weidi Xie. M 3 builder: A multi-agent system for automated machine learning in medical imaging. In *International Workshop on Agentic AI for Medicine*, pp. 115–124. Springer, 2025.

FoundationAgents. Github - foundationagents/metagpt: The multi-agent framework: First ai software company, towards natural language programming · github, February 2025. URL https://github.com/foundationagents/metagpt.

Github. GitHub Copilot · Your AI pair programmer, 2025. URL https://github.com/features/copilot.

Google. Introducing google antigravity, a new era in ai-assisted software development, 2025. URL https://antigravity.google/blog/introducing-google-antigravity.

Google Deepmind. Introducing gemini 3.6 flash, 3.5 flash-lite, and 3.5 flash cyber, July 2026. URL https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-6-flash-3-5-flash-lite-3-5-flash-cyber/.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. Ds-agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453*, 2024.

Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Andrew Hoopes, Neel Dey, Victor Ion Butoi, John V Guttag, and Adrian V Dalca. Voxelprompt: A vision agent for end-to-end medical image analysis. *arXiv preprint arXiv:2410.08397*, 2024.

Ziyan Huang, Haoyu Wang, Jin Ye, Jingqi Niu, Can Tu, Yuncheng Yang, Shiyi Du, Zhongying Deng, Lixu Gu, and Junjun He. Revisiting nnu-net for iterative pseudo labeling and efficient sliding window inference. In *MICCAI Challenge on Fast and Low-Resource Semi-supervised Abdominal Organ Segmentation*, pp. 178–189. Springer, 2022.

Fabian Isensee, Paul F Jaeger, Simon AA Kohl, Jens Petersen, and Klaus H Maier-Hein. nnu-net: a self-configuring method for deep learning-based biomedical image segmentation. *Nature methods*, 18(2):203–211, 2021.

Fabian Isensee, Tassilo Wald, Constantin Ulrich, Michael Baumgartner, Saikat Roy, Klaus Maier-Hein, and Paul F Jaeger. nnu-net revisited: A call for rigorous validation in 3d medical image segmentation. In *International conference on medical image computing and computer-assisted intervention*, pp. 488–498. Springer, 2024.

Sabhay Jain, Xingjian Li, and Min Xu. Knowledge transfer from macro-world to micro-world: enhancing 3d cryo-et classification through fine-tuning video-based deep models. *Bioinformatics*, 40(7):btae368, 2024.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4015–4026, 2023.

Zeki Kuş and Musa Aydin. Medsegbench: A comprehensive benchmark for medical image segmentation in diverse data modalities. *Scientific Data*, 11(1):1283, 2024.

Bennett Landman, Zhoubing Xu, Juan Igelsias, Martin Styner, Thomas Langerak, and Arno Klein. Miccai multi-atlas labeling beyond the cranial vault–workshop and challenge. In *Proc. MICCAI multi-atlas labeling beyond cranial vault—workshop challenge*, volume 5, pp. 12. Munich, Germany, 2015.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. Acecoder: An effective prompting technique specialized in code generation. *ACM Transactions on Software Engineering and Methodology*, 33 (8):1–26, 2024.

Xiangrui Li, Paul S Morgan, John Ashburner, Jolinda Smith, and Christopher Rorden. The first step for neuroimaging data analysis: Dicom to nifti conversion. *Journal of neuroscience methods*, 264: 47–56, 2016.

Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Kay Chen Tan. A survey on evolutionary neural architecture search. *IEEE transactions on neural networks and learning systems*, 34(2):550–570, 2021.

Jun Ma, Yao Zhang, Song Gu, Xingle An, Zhihe Wang, Cheng Ge, Congcong Wang, Fan Zhang, Yu Wang, Yinan Xu, et al. Fast and low-gpu-memory abdomen ct organ segmentation: the flare challenge. *Medical Image Analysis*, 82:102616, 2022.

Jun Ma, Yuting He, Feifei Li, Lin Han, Chenyu You, and Bo Wang. Segment anything in medical images. *Nature Communications*, 15:654, 2024a.

Jun Ma, Yao Zhang, Song Gu, Cheng Ge, Shihao Mae, Adamo Young, Cheng Zhu, Xin Yang, Kangkang Meng, Ziyan Huang, et al. Unleashing the strengths of unlabelled data in deep learning-assisted pan-cancer abdominal organ quantification: the flare22 challenge. *The Lancet Digital Health*, 6(11):e815–e826, 2024b.

Vasilije Markovic, Lazar Obradovic, Laszlo Hajdu, and Jovan Pavlovic. Optimizing the interface between knowledge graphs and llms for complex reasoning, 2025. URL https://arxiv.org/abs/2505.24478.

Peter Mildenberger, Marco Eichelberg, and Eric Martin. Introduction to the dicom standard. *European radiology*, 12(4):920–927, 2002.

OpenAI. Gpt-6 astra system card, September 2026. URL https://deploymentsafety.openai.com/gpt-6-astra.

QwenTeam. Qwen3.8-max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, et al. Sam 2: Segment anything in images and videos. *arXiv preprint arXiv:2408.00714*, 2024.

Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)*, 54(4):1–34, 2021.

Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine-tuning: An empirical assessment of llms for code. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 490–502. IEEE, 2025.

Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. Automl-agent: A multi-agent llm framework for full-pipeline automl. *arXiv preprint arXiv:2410.02958*, 2024.

Jeya Maria Jose Valanarasu, Yucheng Tang, Dong Yang, Ziyue Xu, Can Zhao, Wenqi Li, Vishal M Patel, Bennett Landman, Daguang Xu, Yufan He, et al. Disruptive autoencoders: Leveraging low-level features for 3d medical image pre-training. *arXiv preprint arXiv:2307.16896*, 2023.

Yan Wang, Liangli Zhen, Jianwei Zhang, Miqing Li, Lei Zhang, Zizhou Wang, Yangqin Feng, Yu Xue, Xiao Wang, Zheng Chen, et al. Mednas: Multiscale training-free neural architecture search for medical image analysis. *IEEE Transactions on Evolutionary Computation*, 28(3):668–681, 2024.

Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, and Bingbing Ni. Medmnist v2-a large-scale lightweight benchmark for 2d and 3d biomedical image classification. *Scientific Data*, 10(1):41, 2023.

Fan Zhang, Yu Wang, and Hua Yang. Efficient context-aware network for abdominal multi-organ segmentation. *arXiv preprint arXiv:2109.10601*, 2021.

GuoDong Zhang, WenWen Gu, SuRan Wang, YanLin Li, DaZhe Zhao, TingYu Liang, ZhaoXuan Gong, and RongHui Ju. Motc: Abdominal multi-objective segmentation model with parallel fusion of global and local information. *Journal of Imaging Informatics in Medicine*, 37(3):1–16, 2024.

Li Zhao, Dongming Zhou, Xin Jin, and Weina Zhu. Nn-transunet: an automatic deep learning pipeline for heart mri segmentation. *Life*, 12(10):1570, 2022.

Zixiao Zhao, Jing Sun, Zhe Hou, Zhiyuan Wei, Cheng-Hao Cai, Miao Qiao, and Jin Song Dong. Mactg: Multi-agent collaborative thought graph for automatic programming. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, ICSE '26, pp. 259–271, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400720253. doi: 10.1145/3744916.3773221. URL https://doi.org/10.1145/3744916.3773221.

Alex P Zijdenbos, Benoit M Dawant, Richard A Margolin, and Andrew C Palmer. Morphometric analysis of white matter lesions in mr images: method and validation. *IEEE transactions on medical imaging*, 13(4):716–724, 1994.

A Limitations

Generalisation. MedPilot is evaluated on six representative medical imaging benchmarks covering segmentation and classification across CT and MRI. Although these datasets span different modalities, dimensionalities, and supervision settings, they do not cover the full diversity of medical imaging workflows. Tasks such as registration, detection, denoising and multi-modal fusion may require additional task-specific validators, domain knowledge, and evaluation protocols. Our results should therefore be interpreted as evidence for automated pipeline synthesis within the evaluated task families, rather than as a guarantee of universal coverage across all medical imaging tasks.

Clinical application. MedPilot reduces human intervention through staged validation, but validation does not imply clinical correctness. The current validators check dataset consistency, design compatibility, code execution, module assembly, and benchmark-level performance; they do not replace expert review, external validation, or regulatory assessment. Generated pipelines should therefore be treated as research artefacts for model development and benchmarking, not as clinically deployable systems without further verification.

Context length and bounded refinement. MedPilot accumulates interaction history and validation feedback during iterative refinement. However, its multi-agent decomposition distributes this information across role-specific agent contexts rather than maintaining a single global generation history, reducing the context accumulated by each individual agent. We did not encounter context-window exhaustion in our experiments. Nevertheless, substantially longer workflows or prolonged failure cycles could increase the context of individual agents beyond the regimes evaluated here. The per-stage retry budget ($R_{\max} = 3$) bounds repeated validation-repair cycles, while larger-scale deployments could further employ context summarisation or selective history retention if needed.

Computational and API cost. MedPilot's multi-agent coordination, repeated validation, and iterative repair introduce additional synthesis overhead relative to simpler generation workflows. Across the evaluated agentic baselines, representative end-to-end synthesis times were 27 min for MGX, 54 min for MaCTG, 33 min for AutoML-Agent, 47 min for M^{3}Builder, 22 min for MLZero, and 35 min for MedPilot. Exact token consumption and API cost cannot be compared consistently across all methods because the evaluated systems use heterogeneous execution interfaces, including local frameworks and hosted or IDE-based tools with different levels of usage telemetry. For MedPilot, a complete synthesis costs approximately \$20–\$40 in API usage.

B Implementation Details

B.1 Agent Roles

MedPilot implements pipeline synthesis as a staged multi-agent workflow. The prompts define both router agents, which coordinate stage-level execution, and specialist agents, which perform dataset analysis, architecture design, validation, implementation, and testing. Each prompt specifies the agent's responsibility, accessible tools or knowledge sources, expected artefacts, stopping rules, and feedback-handling behaviour. Table 4 summarises the main agent roles (system prompts) used in our implementation.

Table 4: Agent roles in MedPilot.

Agent	Stage	Responsibility	Main artefact
Planning Router	Planning	Coordinates dataset analysis and architecture design; passes dataset descriptors to the Architect; revises planning outputs using validation feedback.	Planning package
Dataset Analyser	Planning	Performs two-step dataset analysis: first explores folder structure, file counts, formats, and image-label correspondence; then generates and executes a Python analysis script to extract dimensions, spacing, orientation, intensity statistics, label values, and data-quality issues.	`folder_analysis.py`; `folder_analysis_report.txt`; `folder_analysis_data.json`
Architect	Planning	Designs the medical imaging pipeline from dataset analysis, verified system constraints, medical-domain knowledge, and retrieved prior cases; specifies components, interfaces, preprocessing, architecture, training, inference, and hyperparameters.	`architecture_design_nl.txt`; `architecture_design.csp` (rule-based conversion)
Verification agent	Planning validation	Checks whether the design satisfies software-engineering, medical-domain, and specification-completeness criteria, including modality-appropriate preprocessing and task-appropriate metrics. Verifies workflow-level properties of the abstract design through CSP# specifications and PAT, including deadlock freedom, reachability, and safety/liveness properties.	Semantic validation report, `pat_verification_result.txt`
Coding Router	Coding	Parses the validated architecture plan, extracts component specifications, launches component Coders in parallel, coordinates component testing and fix loops, generates the assembly script, and manages final assembly validation.	Component implementation plan
Coder(s)	Coding	Implements Python classes or scripts for medical imaging components according to the validated architecture, including data loading, preprocessing, model definition, training, inference, evaluation, and assembly fixes when validation fails.	Component Python modules (`dataset_module.py`, `neural_network_module.py`, `training_module.py`, `...`); `pipeline_main.py`
Tester	Coding validation	Performs two-phase validation using static analysis and pytest: checks syntax, dependencies, coding quality, component assumptions, functional correctness, interface consistency, and assembly execution; returns structured diagnostics for router-driven fixes.	Test scripts; validation summary

B.2 Agent-Specific Domain Knowledge

MedPilot couples each agent with role-specific domain knowledge rather than providing all agents with a single shared knowledge prompt. This design keeps each context focused: dataset-level agents

receive knowledge about medical image formats and metadata, planning agents receive pipeline-design and software-quality criteria, validation agents receive semantic or formal checking rules, and coding agents receive implementation-level references. Table 5 summarises how domain knowledge is assigned to agents.

Table 5: Agent-specific domain knowledge used by MedPilot.

Agent	Knowledge sources	Role-specific use
Dataset Analyser	Medical image data standards and dataset conventions, including NIfTI, DICOM, NRRD, MHA/MHD, and common image–label organisation patterns	Guides dataset inspection, file-format recognition, metadata extraction, spacing/orientation analysis, image–label matching, label-value inspection, and data-quality checks.
Architect	Medical imaging pipeline practice, including 2D/3D segmentation, 3D classification, semi-supervised learning, modality-aware preprocessing, patch-based training, losses, metrics, and inference conventions	Guides the construction of the abstract pipeline design, including preprocessing, dimensionality choice, architecture family, training strategy, inference procedure, and evaluation protocol.
Verification agent	ISO/IEC 25023:2016 software-quality criteria, modular software-engineering principles, and IEC 62304 medical device software conventions, CSP# process modelling and PAT model checking	Checks whether the proposed design is complete, modular, maintainable, medically appropriate, and consistent with the task specification and dataset characteristics. Checks workflow-level properties such as deadlock freedom, reachability, safety, and liveness.
Coder(s)	MONAI and PyTorch API docs, including data loaders, transforms, network modules, losses, metrics, optimisers, and training utilities	Implements medical imaging pipeline modules using established APIs and coding patterns for data loading, preprocessing, model construction, training, inference, and evaluation.
Tester	Pytest, static analysis, dependency checks, and medical-image-specific testing practice	Generates and executes component tests, checks syntax and dependencies, validates interface consistency, and reports structured diagnostics for failed modules or assembly scripts.

This agent-specific organisation prevents unrelated domain knowledge from overwhelming the context. For example, the Dataset Analyser does not need detailed optimiser or model-architecture references, while the Coder does not need the full semantic-validation rubric. Instead, each agent receives the knowledge needed for its own stage, and validated outputs are propagated through shared memory. This coupling between role and knowledge is central to MedPilot's context-management design.

B.3 Agent Specific Tool and Skill Usage

Tools. MedPilot agents share a common tool layer for file-system access, command-line execution, shared-memory operations, and knowledge retrieval. Specifically, shared memory is implemented using the Eion MCP (Model Context Protocol) server (eiondb, 2025), which stores validated artefacts, validation feedback, execution traces, and intermediate synthesis state across agents. The role-specific and adaptive knowledge bases are implemented using Cognee MCP servers (Markovic et al., 2025), which support retrieval over curated domain knowledge and storage/retrieval of validated prior pipelines. In addition to this common layer, MedPilot uses several specialised tools for validation and code-quality assessment, summarised in Table 6.

Skills. MedPilot uses Agent Skills (Anthropic, 2025) to package recurring multi-step procedures into reusable workflow units. This is important because many operations in pipeline synthesis are not single tool calls, but structured routines with preconditions, intermediate artefacts, validation criteria, and stopping conditions. For example, dataset analysis requires folder inspection, script generation, script execution and report writing; code validation requires static analysis, test generation, and execution. Encoding these routines as skills reduces prompt-level ambiguity and ensures that routers invoke the same procedure consistently across datasets and refinement iterations.

Routers primarily invoke skills, whereas specialist agents execute the concrete tools required by those skills. For example, the Verification agent invokes formal-verification and the Tester invokes static-validation, execution-validation, and assembly-validation skills; these skills in turn call PAT, Pylint, Radon, and Pytest. This separation keeps orchestration logic reusable while preserving specialised validation behaviour for each stage. It also makes the system easier to extend: adding

Table 6: Specialised validation and code-quality tools used by MedPilot.

Tool	Used by	Purpose
PAT	Verification agent	Executes CSP# model-checking specifications generated from the planned workflow. It is used to verify process-level properties such as deadlock freedom, reachability, safety, and liveness before implementation begins.
Pylint	Tester	Performs static code-quality and syntax checks on generated Python modules, helping identify import errors, undefined variables, inconsistent interfaces, and maintainability issues before runtime execution.
Radon	Tester	Measures code complexity and maintainability of generated Python packages. These checks provide an additional signal for overly complex, duplicated, or poorly structured implementation artefacts.
Pytest	Tester	Runs generated component-level and assembly-level tests. It is used to check functional correctness, dependency consistency, input–output contracts, and end-to-end execution of assembled pipeline modules.

a new task family or validator can be done by introducing a new skill without changing the overall agent-routing structure.

B.4 Adaptive Knowledge Base Accumulation

The adaptive knowledge base is populated incrementally during the benchmark sequence. After a pipeline passes all validation stages, MedPilot stores its dataset descriptors, validated architecture specification, component interfaces, implementation artefacts, validation summaries, and task metadata. These artefacts are indexed by task type, modality, dimensionality, dataset structure, preprocessing strategy, architecture family, and evaluation result, allowing later runs to retrieve relevant prior cases without treating them as fixed templates.

In our experiments, datasets are processed in the following order: ACDC, BTCV, FLARE21, FLARE22, Promise12MSBench, and OrganMNIST3D. This order is chosen to support progressive knowledge accumulation across task difficulty and domain coverage. ACDC provides an initial medium-complexity supervised 3D segmentation case with clear structure and well-defined cardiac anatomy. BTCV and FLARE21 then expand the knowledge base to abdominal multi-organ CT segmentation, introducing stronger class imbalance, volumetric memory constraints, and multi-organ preprocessing requirements. FLARE22 further stresses the accumulated knowledge under a more challenging semi-supervised setting. Finally, Promise12MSBench and OrganMNIST3D test whether the accumulated knowledge can be adapted to simpler but structurally different settings, namely 2D binary segmentation and low-resolution 3D classification.

This ordering is not used to hand-engineer dataset-specific solutions. Retrieved cases serve only as references for candidate design choices, and every adapted artefact must pass the same semantic, formal, execution, and assembly validation gates before being committed to shared memory. Thus, the adaptive knowledge base provides a progressively richer prior over pipeline designs while validation prevents inappropriate transfer from earlier tasks.

C Experimental Protocol Details

C.1 Environment

All experiments were run on an Ubuntu 22.04 server with an AMD Ryzen Threadripper PRO 3995WX processor, 256GB RAM, and two NVIDIA RTX A6000 GPUs with 48GB VRAM each. All agent-generated code was executed inside isolated Docker containers to prevent generated scripts from directly affecting the host environment. All methods used containers instantiated from the same base image containing standard medical-imaging and deep-learning libraries, providing a consistent software environment across methods while preserving the command-line and file-system access required for pipeline synthesis and execution. During synthesis, methods were allowed to inspect the container environment and install additional dependencies when required by their generated pipelines.

C.2 Dataset Splits and Evaluation

For each benchmark, the train, validation, and test partitions were fixed before pipeline synthesis and shared across all evaluated methods. The autonomous agents were allowed to generate data-loading and training logic, but they were not permitted to redefine the held-out test set. Model selection and pipeline refinement used only the training and validation partitions, while the fixed test partition was used exclusively for the final downstream evaluation reported in Figure 2. This protocol ensures that all methods are evaluated on identical samples and prevents test-set information from influencing pipeline synthesis.

C.3 Baseline backbone

We apply an aligned LLM configuration of the evaluated systems. MGX was evaluated through its hosted online service; its underlying LLM is not exposed to users and was therefore not controlled in our experiments. MaCTG uses DeepSeek-V3.2 for planning and Qwen3-Coder for execution; AutoML-Agent uses GPT-5.5 Medium; M^3Builder and MLZero use Claude-Opus-4.7 Medium in their reported configurations. MedPilot also uses Claude-Opus-4.7 Medium. Consequently, the comparison evaluates complete systems under their native configurations rather than isolating the effect of agent architecture from the underlying LLM.

C.4 Baseline Prompting

We use different prompting protocols for the three baseline categories, while keeping the task information consistent across all methods. All baselines receive the same task specification, dataset path, output directory, and requirement to construct a trainable and evaluable medical imaging pipeline. The difference lies in how this information is formatted for each framework. All methods were evaluated on the datasets in the same fixed order as MedPilot (ACDC, BTCV, FLARE21, FLARE22, Promise12MSBench, and OrganMNIST3D), ensuring a consistent sequential evaluation protocol across methods. Baseline methods did not receive MedPilot's adaptive knowledge base or any additional cross-task information beyond what is natively supported by their own framework.

Multi-agent coding frameworks. For MGX and MaCTG, we use a four-step construction prompt because these frameworks do not impose a fixed medical-pipeline input schema. The prompt explicitly asks the agents to verify intermediate outputs before proceeding to later stages, so that baselines are also encouraged to perform self-checking within their frameworks rather than one-shot generation.

Step 1: Dataset structure exploration. The method first inspects the dataset using shell commands only. It is instructed to identify the directory tree, file counts, file formats, naming patterns, sample filenames, and image–label correspondence rules. This step forbids Python analysis or pipeline implementation, preventing premature code generation before dataset organisation is understood.

Step 2: Quantitative dataset analysis. The method then generates and executes `folder_analysis.py` to compute dataset-level statistics, including file correspondence, image dimensions, voxel spacing, data types, intensity distributions, label values, class distributions, and data-quality issues. It is instructed to verify that `folder_analysis_report.txt` and `folder_analysis_data.json` are created and contain plausible outputs.

Step 3: Architecture design and component implementation. Using the dataset analysis outputs, the method creates `architecture_design_nl.txt`, a unified natural-language architecture specification covering the dataset module, network architecture, training module, inference module, interfaces, hyperparameters, and workflow transitions. It then implements all components as Python modules following the `*_module.py` naming convention, with requirements for medical-imaging preprocessing, MONAI/PyTorch compatibility, typed signatures, error handling, and executable code.

Step 4: Pipeline assembly. Finally, the method generates `pipeline_main.py` to integrate the generated modules into an end-to-end training and inference workflow. The prompt requires the method to read the actual component files and `folder_analysis_data.json` before assembly, so that the entry point uses real class names, method signatures, dataset paths, split definitions, and component interfaces. The method is also instructed to verify that the assembled entry point is runnable.

Agentic AutoML frameworks. For AutoML-Agent, M^{3}Builder and MLZero, we follow their intended input format. Each method receives the same task specification, dataset path, and output target as the multi-agent baselines, but is additionally given the scaffold required by the corresponding framework: AutoML-Agent receives its required code template, and M^{3}Builder receives its required dataset card and template information. MLZero receives the dataset and task description through its native input workflow, without an externally supplied code template or dataset-specific pipeline scaffold. These inputs are necessary for the frameworks to run according to their published workflow. This setup ensures that all baselines receive the same task content while respecting framework-specific input requirements.

IDE-based coding assistants. GitHub Copilot, Cursor, and Antigravity receive the same four-stage task specification covering dataset analysis, pipeline design, implementation, and assembly. Each assistant is given access to the working directory, dataset, command line, and execution outputs, and is allowed to iteratively revise the generated pipeline in response to runtime feedback.

Because these tools are inherently interactive, all human corrective actions are logged in accordance with the Intervention Burden protocol in Section 3.1. Additional instructions or relayed execution errors are counted as E1 interventions, supplied templates or skeleton code as E2, and manual modifications to generated scripts or configuration files as E3. Self-recovery performed by the assistant without human input is not counted.

D Additional Comparison with nnU-Net

To further contextualise MedPilot against established medical-imaging automation, we compare its segmentation performance with reported nnU-Net results from prior studies on four corresponding benchmarks. OrganMNIST3D is excluded because it is a classification task, whereas nnU-Net is designed for biomedical image segmentation. Promise12 is also excluded because our experiments use Promise12MSBench, a 2D slice-based reformulation, while the available published nnU-Net result is reported on the original volumetric PROMISE12 challenge setting. Because the evaluation settings may differ in data splits, training protocols, and nnU-Net configurations, these results are intended as reference points rather than strictly controlled comparisons.

Table 7: Comparison between MedPilot and reported nnU-Net results on four segmentation benchmarks.

Dataset	nnU-Net	MedPilot	Published nnU-Net evaluation setting
FLARE22	0.8797	0.8094 ± 0.0114	Supervised-only 3D nnU-Net trained on the labelled FLARE22 data (Huang et al., 2022).
ACDC	0.9154	0.8960 ± 0.0077	Original nnU-Net (Isensee et al., 2024).
BTCV	0.8308	0.7619 ± 0.0069	Original nnU-Net (Isensee et al., 2024).
FLARE21	0.8938	0.8691 ± 0.0093	nnU-Net reproduced on FLARE21 (Zhang et al., 2024).

The reported nnU-Net results remain strong across the four segmentation benchmarks, as expected from a framework specialised for biomedical image segmentation. MedPilot achieves performance in the same overall range while addressing a broader problem: it autonomously constructs, validates, and repairs complete medical-imaging pipelines from task and dataset inputs rather than configuring a predefined segmentation framework. The comparison therefore serves as a specialised medical-imaging reference rather than a direct head-to-head baseline.

E Runtime Example: BTCV

This section presents a condensed runtime trace of MedPilot on BTCV. We omit long generated code files and full logs, and instead report the main intermediate prompts, artefacts, validation feedback, and repair actions. The complete generated code is released in the repository.

E.1 INPUT

User request

User: I have a BTCV dataset at `../datasets/BTCV`. Can you generate an entire 3D segmentation pipeline including training and inference for me?

E.2 DATASET ANALYSIS

The Planning Router first decomposes the task into dataset analysis and architecture design. It invokes the Dataset Analyser with the dataset path and task type, and constrains the first step to structure exploration only.

Planning Router → Dataset Analyser: structure exploration

Dataset path: `../datasets/BTCV`
Task type: supervised 3D segmentation

Data analysis substep 1: structure exploration only.

1. Use shell commands to explore the complete directory tree.
2. Identify subdirectories and their purposes.
3. Sample file listings from each directory.
4. Count files in each directory.
5. Analyse naming patterns and file extensions.
6. Detect image–label correspondence rules.

Constraints. Do not write Python scripts, perform detailed image analysis, generate reports, or proceed to the next substep. Stop after summarising the dataset structure.

Dataset Analyser output: structure summary

Folder tree. The Dataset Analyser records the candidate BTCV directory layout, including the dataset root and expected subdirectories: `imagesTr/`, `labelsTr/`, and `imagesTs/`.
File counts and formats. For each discovered directory, the analyser counts files recursively, records the observed file extensions, and stores representative sample filenames. The detected medical-image format is summarised from the observed extensions, typically NIfTI `.nii.gz` for BTCV-style releases.
Naming pattern. The analyser identifies how image and label files should be paired. For standard BTCV layouts, it checks basename matching between `imagesTr/` and `labelsTr/`.

Planning Router → Dataset Analyser: quantitative analysis

Using the discovered folder structure, generate and execute `folder_analysis.py`. The script should compute image dimensions, spacing, orientation, intensity statistics, label values, class distribution, and data-quality checks. It should produce:

- `folder_analysis_report.txt`
- `folder_analysis_data.json`

Stop after verifying that both files are created.

Dataset Analyser output: `folder_analysis.py`

```
...
# Count files and collect sample filenames
image_files = sorted((dataset_root / "imagesTr").glob("*.nii.gz"))
label_files = sorted((dataset_root / "labelsTr").glob("*.nii.gz"))

num_images = len(image_files)
num_labels = len(label_files)
...
# Extract dimensions, spacing, and intensity statistics
dims, spacings, intensity_stats = [], [], []

for img_path in image_files[:20]:
    img = nib.load(str(img_path))
    arr = img.get_fdata(dtype=np.float32)

    dims.append(arr.shape)
    spacings.append(img.header.get_zooms()[:3])
    intensity_stats.append({
        "file": img_path.name,
        "min": float(np.min(arr)),
        "max": float(np.max(arr)),
```

```
        "mean": float(np.mean(arr)),
        "std": float(np.std(arr)),
        "p01": float(np.percentile(arr, 1)),
        "p99": float(np.percentile(arr, 99)),
    })
...
# Count label values and infer number of classes
num_classes = len(label_values)
foreground_classes = [v for v in label_values if v != 0]
...
# Write machine-readable descriptors for the Architect
Path("folder_analysis_data.json").write_text(
    json.dumps(analysis_data, indent=2)
)
...
```

Verification agent: Dataset Analysis Validation

Inputs. `folder_analysis.py`, `folder_analysis_data.json`, and `folder_analysis_report.txt`.
File-level checks. The validator checks that all expected artefacts are present, readable, and non-empty. It also verifies that the JSON file can be parsed and contains the required planning fields, including dataset path, modality, task type, file counts, image–label correspondence, image dimensions, spacing, intensity statistics, and label values.
Semantic checks. The validator checks that the extracted descriptors are medically and technically consistent with BTCV. In particular, it verifies that the task is recognised as supervised 3D abdominal CT segmentation, that the expected NIfTI image and label layouts are handled, that the inferred pairing rule is valid, and that the reported label values and class count are compatible with multi-organ segmentation.
Outcome. The dataset-analysis artefacts pass validation and are committed to shared memory. The Architect can then condition on the validated descriptors rather than raw folder observations.

E.3 Architecture Design

After dataset-analysis validation succeeds, the Planning Router invokes the Architect with the validated descriptors from shared memory. This prevents the architecture design stage from conditioning on incomplete or unverified dataset observations.

Planning Router → Architect

Design a complete supervised 3D CT segmentation pipeline using the validated BTCV dataset analysis. The design should specify data loading, preprocessing, augmentation, model architecture, loss functions, training configuration, inference strategy, evaluation metrics, component interfaces, and workflow state transitions. Output a unified natural-language architecture specification in `architecture_design_nl.txt`.

Table 8: Initial BTCV architecture draft produced by the Architect.

Component	Initial design decision
Dataset module	MONAI dictionary-style dataset
Preprocessing	CT intensity normalisation, foreground cropping, fixed-size resizing
Sampling	Patch-based training
Augmentation	Rotation, flipping, noise, intensity jitter
Model	3D U-Net-style segmentation network
Loss	Dice + cross-entropy
Inference	Sliding-window inference
Evaluation	Per-class + mean foreground Dice

Verification agent: Architecture Semantic Validation

Status: fail.
Issue. The preprocessing specification uses fixed-size resizing for BTCV volumes. This is inappropriate for 3D CT segmentation because resizing changes voxel geometry without explicitly preserving physical spacing. For anisotropic medical volumes, downstream patch extraction, model receptive field, and output reconstruction should be defined in voxel/physical space rather than by image-array size alone.
Required repair. Replace fixed-size resizing with spacing-aware resampling. The preprocessing component should explicitly specify canonical orientation handling, voxel-spacing normalisation, CT intensity clipping/normalisation, and consistent spatial assumptions for training and sliding-window inference.

Architect repair action

The Architect revises `architecture_design_nl.txt` by removing fixed-size resizing and adding spacing-aware resampling before patch extraction. The revised design propagates the target spacing and orientation assumptions to the dataset, preprocessing, and inference components, then resubmits the design for validation.

After semantic validation passes, the verified natural-language design is converted into a CSP# workflow specification by a fixed rule-based script. Although the design is expressed in natural

Table 9: Revised BTCV architecture decisions after semantic validation.

Component	Validated design decision
Dataset module	MONAI dictionary-style dataset
Preprocessing	Orientation normalisation, spacing-aware resampling, CT intensity normalisation
Sampling	Foreground-aware patch-based training
Augmentation	Rotation, flipping, noise, intensity jitter
Model	3D U-Net-style segmentation network
Loss	Dice + cross-entropy
Inference	Sliding-window inference with spacing-aware output reconstruction
Evaluation	Per-class + mean foreground Dice

language, its structure is constrained by the Architect prompt to adhere to the predefined component and transition fields required by the rule-based translator. The generated specification models pipeline components as concurrent processes and tracks dataflow readiness through explicit state variables.

Converted CSP# from the validated BTCV design

```
// Component states and BTCV pipeline modules
#define STATE_INACTIVE 0;
#define STATE_PROCESSING 2;
#define STATE_READY 3;
#define STATE_ERROR 4;

#define DATASET_MODULE 0;
#define NEURAL_NETWORK 1;
#define TRAINING_MODULE 2;
#define INFERENCE_MODULE 3;

var componentStates[4] =
  [STATE_INACTIVE, STATE_INACTIVE,
   STATE_INACTIVE, STATE_INACTIVE];
...
// Data-flow flags
var raw_data_available = 1;
var preprocessed_data_ready = 0;
var model_initialized = 0;
var trained_model_ready = 0;
var results_saved = 0;
...
// Guarded transition: preprocessing must finish
// before downstream training can begin.
DatasetModule =
  [componentStates[0] == STATE_PROCESSING]
  normalize_intensity{
    componentStates[0] = STATE_READY;
    preprocessed_data_ready = 1;
  } -> DatasetModule();

// Guarded transition: training starts only after
// both model initialisation and preprocessing are complete.
TrainingModule =
  [componentStates[2] == STATE_INACTIVE &&
   model_initialized == 1 &&
   preprocessed_data_ready == 1]
  initialize_trainer{
    componentStates[2] = STATE_PROCESSING;
  } -> TrainingModule();
...
// System composition
BtcvMultiOrganCtSegmentationSystem =
  DatasetModule() ||| NeuralNetwork() |||
  TrainingModule() ||| InferenceModule();
...
// Verification predicates
#define training_finished (trained_model_ready == 1);
#define inference_complete (results_saved == 1);
#define no_errors
  (componentStates[0] != STATE_ERROR &&
   componentStates[1] != STATE_ERROR &&
   componentStates[2] != STATE_ERROR &&
   componentStates[3] != STATE_ERROR);
...
// Assertions
#assert BtcvMultiOrganCtSegmentationSystem deadlockfree;
```

```
#assert BtcvMultiOrganCtSegmentationSystem |= <> training_finished;
#assert BtcvMultiOrganCtSegmentationSystem |= <> inference_complete;
#assert BtcvMultiOrganCtSegmentationSystem |= [] no_errors;
```

Verification Agent → PAT

Checks. PAT verifies deadlock freedom, reachability of preprocessing completion, reachability of training completion, reachability of inference completion, and the safety property that no component enters `STATE_ERROR`.
Outcome. The BTCV workflow passes formal verification. The validated architecture is committed to shared memory and passed to the Coding Router for component-level implementation.

E.4 MODULE CODING

After the architecture specification passes semantic and formal validation, the validated design is committed to shared memory and passed to the Coding Router. The Coding Router decomposes the design into component-level implementation tasks and launches specialised Coders.

Coding Router → Coders

Input. Validated `architecture_design_nl.txt`, dataset descriptors from `folder_analysis_data.json`, and verified workflow constraints from the planning phase.
Component tasks. Generate the following modules using the validated component interfaces:

- `dataset_module.py`: BTCV data loading, preprocessing, augmentation, and patch sampling.
- `neural_network_module.py`: 3D segmentation network.
- `training_module.py`: supervised training, validation, checkpointing, and Dice-based metrics.
- `inference_module.py`: sliding-window inference, post-processing, and prediction export.

Constraint. Each component must follow the method signatures and hyperparameters specified in the validated architecture design. Coders stop after creating their assigned modules and do not perform validation themselves.

Coding Router → Parallel Coders

Parallel assignment. The router then assigns independent implementation tasks to specialised Coders:

- Coder (dataset): implement data loading, preprocessing, augmentation, and patch sampling.
- Coder (neural network): implement the 3D segmentation architecture.
- Coder (training): implement training, validation, checkpointing, and metric computation.
- Coder (inference): implement sliding-window inference, post-processing, and prediction export.

Output contract. Each Coder writes one component module following the `*_module.py` naming convention and the interfaces specified in the validated architecture. The Coders stop after module generation; testing and repair are handled in the subsequent validation loop.

After all component Coders finish their assigned modules, the Coding Router does not directly call for pipeline assembly. Instead, the Tester is invoked for two-phase component validation. In the first phase, the Tester performs static validation, including syntax checks, and code-quality analysis. In the second phase, it generates and executes functional tests from the validated architecture specification and the generated module interfaces, so that tensor-shape errors, input–output contract mismatches, and runtime failures can be detected before assembly.

Tester: static component validation

The Tester first invokes Pylint and Radon on the generated Python modules. Pylint checks syntax, imports, unresolved names, and coding errors, while Radon measures code complexity and maintainability. Modules that fail these checks are returned to the Coding Router with structured diagnostics before any functional tests are executed.

Tester: functional test generation and execution

Input. The Tester receives the statically valid modules and their expected component interfaces.
Functional tests. The Tester creates component-level tests for input–output contracts and minimal execution. For the BTCV training module, one of the generated tests instantiates a mock MONAI-style batch and checks whether the training step can consume the dataset output format:

```
batch = {
    "image": torch.randn(1, 1, 96, 96, 96),
    "label": torch.randint(0, 14, (1, 96, 96, 96)),
}
loss = trainer.train_step(batch)
assert torch.is_tensor(loss)
```

Tester feedback: component-level validation failure

Status: fail.
Issue. The dataset module returns a batch dictionary with keys `"image"` and `"label"`, while the training module initially expects tuple-style batches `(image, label)`. This causes the training loop to fail during batch unpacking.
Failed assumption.

```
images, labels = batch
```

Required repair. Update the training loop to consume MONAI-style dictionary batches and move both tensors to the target device.

Coder repair action

```
images = batch["image"].to(self.device)
labels = batch["label"].to(self.device)
```

The repaired training module is resubmitted to the Tester for static and execution validation.

E.5 ASSEMBLY

After all component modules pass static and functional validation, the Coding Router starts the assembly stage. Unlike component generation, assembly requires the Coder to read the actual generated module files, because class names, constructor arguments, and method signatures may differ from the abstract plan after repair.

Coding Router → Coder: assembly generation

Input. Validated component modules, `folder_analysis_data.json`, and the repaired architecture specification.
Assembly task. Generate `pipeline_main.py`, an executable entry point that integrates the dataset, model, training, and inference components into a complete BTCV workflow.
Required inspection. Before writing the assembly script, the Coder must read:

- `dataset_module.py`
- `neural_network_module.py`
- `training_module.py`
- `inference_module.py`
- `folder_analysis_data.json`

This ensures that `pipeline_main.py` uses actual class names, constructor signatures, dataset paths, split information, and component interfaces rather than placeholders.

Coder → `pipeline_main.py`

```
...
config = load_config("folder_analysis_data.json")
data_root = config["dataset_root"]
...
train_dataset = BTCVDataset(
    data_root=data_root,
    split="train",
    transform=train_transforms,
)
...
model = BTCVUNet3D(
    in_channels=1,
    out_channels=14,
)
...
trainer = BTCVTrainer(
    model=model,
    train_dataset=train_dataset,
    val_dataset=val_dataset,
    device=device,
)
trainer.fit()
...
```

Tester → Assembly validation

Checks. The Tester validates that `pipeline_main.py` imports all generated modules, instantiates each component with compatible arguments, uses dataset paths from `folder_analysis_data.json`, and exposes an executable training/inference entry point.
Outcome. The assembled BTCV pipeline passes validation. The final codebase is committed as a validated artefact and becomes eligible for storage in the adaptive knowledge base.