Kyiv School of Economics

Computer Science Department

Software Engineering & Business Analysis

Bachelor's Thesis

Local LLM-Based AppleScript Generation for macOS Automation

Sofiia Mazepa

Supervisor
KSE, Roman Mishchenko, rmishchenko@kse.org.ua

Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited within the body of this work.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

Place, date	Kyiv, 03.06.2026
Signature	_____

Contents

Abstract

Code generation with large language models is well-studied for mainstream languages (Chen et al., 2021 Jimenez et al., 2024). Still, low-resource programming languages, those that are underrepresented in pretraining corpora and lack established evaluation benchmarks, receive little attention (Joel et al., 2024 Cassano et al., 2024). The problem intensifies when generation must happen on a small model deployed locally, under strict memory and latency constraints. This thesis addresses this gap by using AppleScript, a macOS automation language for which no public datasets or established benchmarks exist, and by examining per-application scripting interfaces that are not generalizable.

The work produces four artifacts: a validated dataset of executable AppleScript tasks spanning 13 macOS applications with an automated virtual-machine-based evaluation pipeline, a comparative study of five fine-tuned open-weights architectures against three frontier cloud models, a macOS application that integrates the selected model behind a multi-stage pipeline with a safety architecture addressing the risks of executing LLM-generated code on a user's machine, and an empirical characterisation of the privacy-versus-quality trade-off, demonstrating that local models do not yet match cloud alternatives on code-generation reliability but offer viable deployment under privacy, cost, and offline constraints.

Keywords:

Large Language Models, *Code Generation*, *AppleScript*, *Low-Resource Programming Languages*, *Parameter-Efficient Fine-Tuning*, *On-Device Inference*, *macOS Automation*

1 Introduction

Code generation has emerged as one of the most consequential applications of large language models, with the academic literature (Chen et al., 2021 Jimenez et al., 2024) and industry adoption tracking its rapid maturation. On mainstream programming languages such as Python, JavaScript, and Java, frontier cloud models—including GPT-5.5, Claude Opus 4.7, and Gemini 3.1 Pro—now exceed 80% on SWE-Bench Verified (Jimenez et al., 2024), a benchmark of real-world GitHub issues that has become the field's de facto standard for measuring practical coding ability. An entire generation of developer tools, including Claude Code, Cursor, Windsurf, Codex, and GitHub Copilot, has been built around routing code-generation requests to these cloud systems, reflecting the convergence of economic, latency, and reliability requirements on a cloud-first deployment model for code-generation LLMs.

In parallel, smaller open-weights architectures distributed through runtimes such as Ollama, LM Studio, and MLX have begun to deliver competent general-purpose performance on consumer hardware (Abdin et al., 2024 Dubey et al., 2024). Their adoption has been motivated by privacy preservation, the absence of per-request inference costs, and freedom from network dependencies, and they now perform competitively on summarisation, translation, classification, and dialogue tasks. For code generation, however, local models remain a distinctly secondary option: smaller models consistently underperform on the joint requirement of syntactic and semantic correctness that code generation imposes, and the gap widens further when the target language is itself underrepresented in pretraining corpora. The literature has begun to formalize this category as Low-Resource Programming Languages (LRPLs; Cassano et al., 2024), with recent work showing substantial performance degradation on such languages compared to high-resource analogs even for code-specialized models (Giagnorio et al., 2025).

This thesis investigates whether a locally deployed language model can reliably translate natural-language instructions into executable AppleScript, macOS's native automation language, and a representative LRPL for which neither benchmarks nor public training data previously existed. AppleScript exposes the scripting interface of nearly every macOS system component and built-in application. However, its adoption has been limited by both its English-like syntax, which is unfamiliar to programmers trained in mainstream languages, and the per-application variability of its scripting dictionaries. A reliable natural-language-to-AppleScript translation layer would, in principle, allow macOS users to access AppleScript's capabilities without prior expertise in the language, provided such a system can operate within the resource constraints of consumer hardware.

2 Research

The work undertaken in this thesis sits at the intersection of three actively researched areas: large language model code generation, low-resource programming languages, and on-device inference on consumer hardware. This chapter reviews each area in relation to the central research question and identifies the specific gap the thesis addresses.

2.1 Code generation with large language models

Automated code generation has become one of the most actively studied applications of large language models, and both the academic literature (Chen et al., 2021 Jimenez et al., 2024) and industry practice reflect its rapid maturation. As of mid-2026, the strongest cloud-hosted systems (Claude Opus 4.7, GPT-5.5, Gemini 3.1 Pro) all exceed 80% on SWE-Bench Verified (Jimenez et al., 2024), the de facto industry standard benchmark for measuring real-world coding capability on Python repositories (BenchLM, 2026).

In parallel, a generation of small open-weights models (Phi-4 (Abdin et al., 2024), Gemma 3, Qwen 3 (Qwen Team, 2025), Llama 3.3 (Dubey et al., 2024), Mistral Small 3) distributed through runtimes such as Ollama, LM Studio, and MLX has reached general-purpose performance on standard benchmarks that was characteristic of frontier cloud models two to three years earlier. Code generation, however, makes more demanding requirements than most natural-language tasks: models must enforce syntactic and semantic correctness simultaneously and internalize structural constraints that token-level prediction alone cannot guarantee. Smaller models underperform consistently on these dimensions (Joel et al., 2024), and the gap widens further on tasks involving languages or libraries that are underrepresented in pretraining corpora, which is the regime in which this thesis operates.

2.2 The low-resource programming language problem

The code generation literature has formalized the category of Low-Resource Programming Languages (LRPLs): languages that are sparsely represented in pretraining corpora, lack standardized evaluation benchmarks, and consequently exhibit substantially worse generation quality than mainstream languages, even from frontier models (Joel et al., 2024 Cassano et al., 2024). The survey by Joel et al. catalogs the dominant failure modes: models transpose idioms from high-resource languages such as Python or JavaScript onto the target syntax, invent functions and APIs that do not exist, and fail to respect language-specific structural constraints.

AppleScript falls into this category and is additionally a domain-specific language tied to a particular operating system. Although it has served as the native automation layer of macOS since 1993, it appears only sparsely in publicly available code corpora: The Stack v2 (Lozhkov et al., 2024), the dataset behind StarCoder2, includes AppleScript with substantially fewer files than shell scripting languages such as Bash or PowerShell. Two language-specific properties make the difficulty greater. First, AppleScript's English-like syntax, designed for human readability rather than parser regularity, introduces lexical ambiguity that more formally specified languages avoid. Second, each scriptable appli-

cation exposes its own scripting interface through an XML-based Scripting Definition (`.sdef`) file. Hence, the set of valid commands, classes, and properties varies from one application to the next: a syntactically correct script for Calendar may be invalid or semantically incoherent when applied to Reminders. Even frontier models exhibit the failure modes cataloged by Joel et al. when generating AppleScript, including Python-style control flow applied to AppleScript constructs and failure to wrap commands in the application-scoped `tell` blocks that the language requires.

2.3 Existing AppleScript resources

No consolidated, validated dataset of AppleScript commands exists in the research community. The available resources are distributed across individual developer repositories on GitHub, generic code corpora such as The Stack v2 (Lozhkov et al., 2024) that include AppleScript as one of many minority languages, application `.sdef` schemas that describe what can be written rather than how, and dispersed community forums (MacScripter, Stack Overflow, Apple's developer documentation). None of these sources provides the combination of natural-language descriptions, parameter schemas, and execution validation that supervised fine-tuning of a code-generation model requires.

The contrast with mainstream languages is sharp: HumanEval (Chen et al., 2021) provides 164 hand-curated Python problems with function signatures, docstrings, and unit tests, and equivalent or larger benchmarks now exist for Java, C++, Rust, Go, JavaScript, and TypeScript through MultiPL-E (Cassano et al., 2022) and SWE-Bench (Jimenez et al., 2024). The next chapter describes the construction of a validated AppleScript dataset as one of the contributions of this thesis, motivated specifically by the absence of a comparable resource.

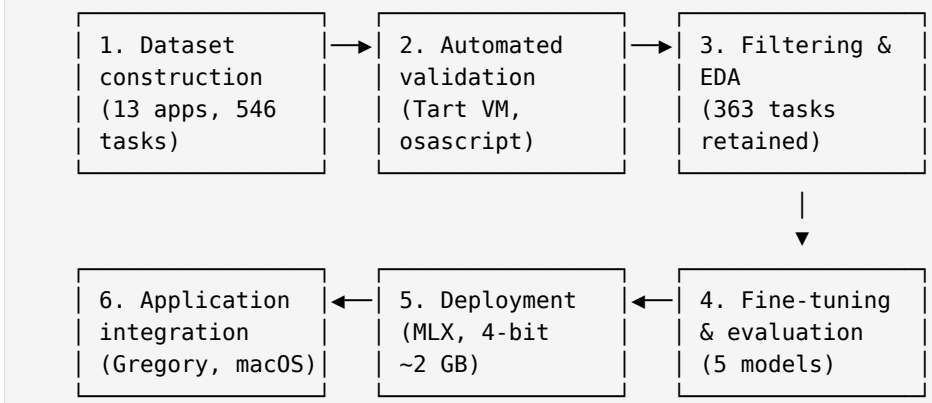
2.4 Apple Intelligence and Siri as quality reference points

Apple Intelligence and Siri provide the most informative reference points, since they, too, operate on-device using a small model (approximately 3 billion parameters; Gunter et al., 2024 Apple, 2025) and target natural-language control of the operating system. Apple Intelligence parses natural language to identify user intent. Then it matches that intent against pre-registered App Intents, which are schemas supplied by application developers that specify the inputs and outputs of supported actions. Natural-language requests route to the registered handlers and are assembled into action pipelines through tool calling. This design avoids the LLM code-generation problem entirely by delegating the action surface to developer-defined schemas. The trade-off is that any action not registered as an App Intent remains unreachable through the natural-language interface, and Apple Intelligence cannot extend itself to applications whose developers have not opted in. A system that generates AppleScript directly, as this thesis proposes, addresses the complementary case: it covers every scriptable application without requiring per-application developer integration, at the cost of accepting the lower reliability inherent in current local code generation.

3 Design

3.1 System Architecture

The work described in this thesis comprises an offline pipeline that builds the dataset, fine-tunes the candidate models, and produces deployment artifacts, together with a runtime application that consumes these artifacts to serve user requests via a natural-language interface. This section provides an overview before subsequent chapters describe each stage in detail.



Listing 1: End-to-end offline pipeline scheme

Offline pipeline. No stage begins until the verified output of the preceding stage is available.

1. **Stage 1** generates synthetic AppleScript tasks grounded in each application's scripting dictionary.
2. **Stage 2** executes each task in an isolated macOS virtual machine and records its success.
3. **Stage 3** filters the results down to the 363 successfully executed tasks that constitute the training corpus.
4. **Stage 4** fine-tunes five candidate language models on that corpus and evaluates them on a held-out test split.
5. **Stage 5** converts the winning model's LoRA adapter into a 4-bit MLX checkpoint suitable for on-device inference and parameterizes the validated scripts into a reusable cached library.
6. **Stage 6** integrates both deployment artifacts into a macOS application that exposes the entire pipeline to end users through a natural-language interface.

```
User input
   │
   ├─▶ Stage 1   Conversational classifier      gpt-4o-mini
   │           (chitchat, capability queries, greetings)
   │
   ├─▶ Stage 2   Safety guardrail               keyword + gpt-4o-
mini
   │           (refuse harmful / destructive intent)
   │
   ├─▶ Stage 3   Cached AppleScripts            gpt-4o
   │           (semantic match against 362 validated templates)
   │
   ├─▶ Stage 4   System-settings commands       gpt-4o-mini
   │           (185 deterministic macOS settings entries)
   │
   └─▶ Stage 5   On-device Phi-3.5 (MLX)        local subprocess
                (fresh AppleScript generation from fine-tuned model)
                   │
                   ▼
            ScriptComposer → ScriptExecutor → result
```

Listing 2: Runtime request pipeline

Runtime architecture. The application routes each user request through a five-stage cascade that consumes the deployment artifacts produced by Stage 5: the quantized model and the cached scripts library. The cascade orders its handlers by cost and reliability. The cheapest and most deterministic handlers come first, and the local generative model, the most expensive and least reliable component, runs only when none of the earlier stages can serve the request. This ordering is the central architectural decision of the application. The absolute reliability limit of a small fine-tuned model on a low-resource language never bounds user-visible quality for most in-distribution requests, as the deterministic stages handle this before the generator runs.

Two deployment modes. The application supports on-device operation (default) and cloud-assisted operation. Both modes run the same five-stage cascade in the same order; only Stage 5 differs. On-device mode serves Stage 5 from the local Phi-3.5-mini model via MLX, so no data leaves the machine. Cloud-assisted mode routes Stage 5 to GPT-4o-mini via the OpenAI API. The choice persists in user preferences and can be toggled at any time. Because Stages 1 through 4 are identical in both modes, the divergence at Stage 5 affects only those requests that none of the deterministic stages can serve.

3.2 Dataset Construction

3.2.1 Motivation and design rationale

The absence of any existing benchmark for AppleScript generation is the defining constraint of this work. AppleScript does not appear in HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), MultiPL-E (Cassano et al., 2022), BigCodeBench (Zhuo et al., 2024), or SWE-Bench (Jimenez et al., 2024). The dataset must therefore be constructed from scratch, and every downstream stage depends on its quality.

This situation is not unique: the low-resource programming language literature documents analogous gaps for languages such as Racket, Lua, and domain-specific scripting languages (Joel et al., 2024 Cassano et al., 2024). AppleScript poses an additional challenge that generic LRPLs do not: every scriptable macOS application exposes its own scripting dictionary, and these dictionaries vary between applications. A model that learns to script Calendar cannot generalize to Reminders without explicit exposure. The dataset must therefore cover multiple applications with sufficient per-application depth, not just overall volume.

The construction pipeline addresses these constraints through three complementary acquisition strategies: sdef-grounded synthetic generation (the primary source), cross-language translation from Bash, and aggregation of existing open-source AppleScript code. The three streams merge and undergo execution-based filtering on a real macOS virtual machine before the final corpus is assembled.

3.2.2 Scripting dictionary extraction

The authoritative specification of what each macOS application supports programmatically is its Scripting Definition file. Every scriptable macOS application ships with an sdef file: an XML document enumerating the application's scripting suites, classes, commands, properties, enumerations, and containment hierarchies.

sdef files are extracted from the native application bundles, which return the merged dictionary for any scriptable application. The pipeline covers 22 applications: Automator, Bluetooth File Exchange, Calendar, Contacts, Finder, Google Chrome, Keynote, Mail, Messages, Music, Notes, Numbers, Pages, Photos, Preview, QuickTime Player, Reminders, Safari, Script Editor, Shortcuts, System Settings, and Terminal.

Grounding generation in the sdef directly addresses the per-application variability challenge by providing the model with the authoritative interface specification at generation time and constraining the output to real commands and properties rather than hallucinated interfaces.

3.2.3 Synthetic generation from scripting dictionaries

The primary data source is a synthetic generation pipeline that uses Gemini 2.5 Flash as the generation model. For each target application, the pipeline provides the model with the parsed sdef excerpt and a structured prompt specifying the desired task complexity (intermediate, advanced, or expert), category (professional, everyday, niche, or academic), and a seed natural-language request. The model generates both a plausible user command

and a corresponding AppleScript implementation that adheres to the sdef-defined interface.

Each generated record contains six fields: a unique identifier, the target application name, the natural-language user request, a complexity classification, a category label, a technical-logic description explaining the implementation approach, and the generated AppleScript code. This structure is preserved unchanged through all subsequent pipeline stages and forms the schema of the final training corpus.

3.2.4 Cross-language translation from Bash

The second source leverages the observation that many macOS system automation tasks have functional equivalents in Bash and other shell scripting languages. Following the MultiPL-T methodology (Cassano et al., 2024), the pipeline identifies Bash automation scripts whose operations overlap with AppleScript's capabilities (filesystem operations, application control, system configuration). It translates them into syntactically valid AppleScript via Gemini 2.5 Flash.

The translation prioritizes functional equivalence over literal correspondence. A Bash script that creates a directory and populates it with a file translates into the equivalent Finder AppleScript commands that produce the same filesystem state through the application's scripting interface, rather than a `do shell script` wrapper that would forward to the original shell command.

3.2.5 Open-source aggregation

The third source aggregates existing AppleScript code from public GitHub repositories and community forums (MacScripter, Stack Overflow). This collection of 84 tasks provides real-world usage patterns that do not arise from synthetic generation alone: idiomatic error-handling constructions, multi-application workflows, and defensive coding patterns. Collected snippets are normalized to the common schema and undergo the same execution-based filtering as the generated tasks.

3.2.6 Setup and validation script generation

Execution-based filtering requires two auxiliary artifacts for each task: a setup script that creates any prerequisite state the task assumes, and a validation script that verifies the task's postconditions after execution.

Setup scripts. Many automation tasks presuppose objects that would not be present in a clean virtual machine snapshot. A task that renames a Reminders list, for instance, requires that list to exist before the main script runs. For each task, Gemini 2.5 Flash receives the task description, the generated AppleScript, and the relevant sdef excerpt, and determines whether a prerequisite state is needed. If so, it generates an idempotent setup script in the most appropriate language: Bash for filesystem operations (file creation, directory structures, permission setting), AppleScript for application-managed data (Reminders lists, Calendar events, Notes, Contacts entries). Idempotency is enforced by requiring existence checks before object creation. Tasks that create everything from scratch are flagged as not requiring setup.

Validation scripts. Every task receives a validation script regardless of whether setup is required. Validation scripts run after the main task script to verify that the task was carried out correctly. Bash validation scripts exit with code 0 for success and non-zero for failure; AppleScript validation scripts return exactly the string **`"true"`** on success and **`"false"`** on failure. The language choice follows the same heuristic as setup scripts. For read-only or display tasks with no verifiable postcondition, a trivial **`exit 0`** check is generated with an appropriate annotation. This produces a consistent three-phase execution structure for every task: optional setup, main execution, and mandatory validation.

3.2.7 Execution-based validation on a macOS virtual machine

To verify functional correctness, each generated script is executed on a real macOS virtual machine. Syntactic validity alone is insufficient: a script may parse correctly yet fail at runtime due to incorrect property references, missing application objects, or GUI timing issues.

The testing infrastructure is built on Tart, an open-source virtualization tool that uses Apple's Virtualization framework to run macOS virtual machines with near-native performance on Apple Silicon. Tart provides OCI-compatible image distribution for reproducible environments and snapshot/clone capabilities essential for deterministic test isolation. The VM is configured with SSH access enabled, a standard user account with passwordless sudo, and Accessibility permissions granted to the Terminal process for UI scripting.

Communication between the host pipeline and the VM occurs over SSH with key-based authentication. Each task is processed through five deterministic stages:

1. **Environment reset.** Application-specific cleanup scripts remove residual state from the previous task (Calendar deletes all calendars; Reminders removes all lists; Finder clears Desktop and Documents).
2. **Setup execution.** If a setup script exists, it runs first. Setup failures are logged, but do not abort the main task.
3. **Main script execution.** The generated AppleScript is piped to **`osascript`** on the VM. Exit code, stdout, and stderr are captured.
4. **Validation execution.** The validation script runs, and its result (exit code in Bash, return value in AppleScript) is recorded.
5. **Result logging.** Success/failure, validation outcome, and raw output are written to a JSON file.

The testing batch covered 13 applications across 644 tasks, with an overall execution success rate of 80.75% (520 tasks). Application-level pass rates ranged from 50.9% (Automator, which requires GUI scripting) to 92.2% (Terminal, which operates through well-defined shell primitives). Validation pass rates reached 39.97%, reflecting the conservative validation contract: only tasks with unambiguous, machine-verifiable postconditions were checked.

Application	Tasks	Passed	Pass rate
Terminal	51	47	92.2%
Contacts	47	43	91.5%
Notes	49	43	87.8%
Safari	48	42	87.5%
Shortcuts	46	40	87.0%
Calendar	50	42	84.0%
Script Editor	49	41	83.7%
Reminders	63	52	82.5%
QuickTime Player	45	36	80.0%
Finder	72	57	79.2%
System Settings	20	15	75.0%
Preview	47	33	70.2%
Automator	57	29	50.9%

Table 1: Execution-based validation results per application

3.2.8 Dataset assembly and final statistics

The final training corpus is assembled by merging three streams, deduplicated by task ID: 363 tasks that passed VM execution in the current pipeline, 216 from an earlier validated pipeline run, and 84 from an open-source aggregation. After deduplication, the combined dataset contains 546 tasks distributed across all application domains.

The corpus is diverse along two independent dimensions. Complexity spans three tiers, and the category distribution covers four use-case types. This deliberate stratification exposes fine-tuned models to a range of task difficulties and use cases, reducing the risk of overfitting to a single automation pattern.

Dimension	Value	Count	Share
Complexity	Intermediate	289	44.9%
	Advanced	236	36.6%
	Expert	119	18.5%
Category	Professional	310	48.1%
	Everyday	192	29.8%
	Niche	135	21.0%
	Academic	7	1.1%

Table 2: Final dataset distribution by complexity and category

3.3 Model Selection and Fine-Tuning

3.3.1 Candidate model selection

Five open-weight transformer models were selected for fine-tuning, spanning a representative range of parameter counts, base architectures, and pre-training corpora relevant to code generation.

Model	Parameters	Architecture	Primary pre-training focus
Llama-3.2-1B-Instruct	1B	Llama 3 decoder	General instruction following
Phi-3.5-mini-instruct	3.8B	Phi-3 decoder	Reasoning + code
Gemma-2-2B-it	2B	Gemma 2 decoder	Multilingual, code
StarCoder2-3B	3B	StarCoder2 decoder	Code (80+ languages)
Mistral-7B-Instruct-v0.3	7B	Mistral decoder	General + instruction

Table 3: Fine-tuning candidate models.

The selection deliberately includes StarCoder2 (Lozhkov et al., 2024), which has some AppleScript representation in The Stack v2, alongside general-purpose instruction models that have little or none. This contrast tests whether code-specialized pre-training provides a meaningful advantage over domain adaptation through fine-tuning alone. The 1B-7B parameter range reflects practical deployment constraints: the target inference environment is an on-device macOS application, where models must run efficiently on Apple Silicon without cloud connectivity.

3.3.2 Parameter-efficient fine-tuning with LoRA

The five candidate models range from 1B to 7B parameters, and full fine-tuning at that scale would exceed the memory budget of the consumer Apple Silicon hardware targeted by the project. Low-Rank Adaptation (LoRA) (Hu et al., 2021) addresses this by freezing the pre-trained weights and injecting trainable rank-decomposition matrices into a subset of the model's linear layers. Only the adapter parameters are updated during training; the base model weights remain unchanged. This reduces the number of trainable parameters by two to three orders of magnitude relative to full fine-tuning, preserving the model's general capabilities and reducing the risk of catastrophic forgetting. All five models share the LoRA configuration shown in Table 3.4.

Hyperparameter	Value
Rank r	16
Scaling factor α	16
Dropout	0.05
Bias	none
Target modules	all linear projections
Task type	CAUSAL_LM

Table 4: Shared LoRA configuration.

3.3.3 Training data split

The combined 546-task dataset is partitioned into training, validation, and test subsets using a stratified split that preserves the joint distribution of application, complexity, and category across all three subsets. Groups with fewer than three tasks are allocated entirely to training to prevent rare combinations from contaminating the validation or test sets.

The split ratios are 80% / 10% / 10%, yielding approximately 382 training tasks, 84 validation tasks, and 80 test tasks. The test split is serialized once, and the evaluation pipeline cross-checks against it using a deterministic seed to guarantee that no test task is seen during training. The test set contains 40 intermediate-complexity, 25 advanced, and 15 expert tasks, distributed across professional (34), everyday (30), and niche (16) categories.

3.3.4 Training configuration and infrastructure

All five adapters are trained with the HuggingFace TRL SFTTrainer for supervised fine-tuning (SFT) under the shared configuration in Table 3.5.

Hyperparameter	Value
Maximum sequence length	4 096 tokens
Per-device batch size	1
Gradient accumulation steps	8 (effective batch = 8)
Number of epochs	3
Learning rate	2×10^{-4}
LR scheduler	Cosine with warmup ($\approx$ 20% of steps per epoch)
Optimiser	AdamW
Weight decay	0.01
dtype	bfloat16
Sequence packing	disabled
Random seed	3 407

Table 5: Shared SFT training configuration.

Sequence packing is explicitly disabled because combining multiple examples into a single sequence can corrupt completion-only masking by placing the label boundary at an

ambiguous position when a completion token from one example appears mid-sequence. Each training example is padded independently to the maximum length in its batch.

Training is conducted on a Google Colab instance equipped with an NVIDIA A100 GPU. All five models are trained and evaluated on this hardware without modification.

3.3.5 Inference prompt format

At inference time, each model receives the same prompt structure used during training, minus the assistant turn, with **`add_generation_prompt=True`** to trigger completion. The format is model-family-specific at the chat-template level but semantically identical across all models. Gemma-2, which lacks native multi-turn system-role support, receives the system prompt prepended to the user message. StarCoder2, which is not instruction-tuned, receives a raw **`### System:`** / **`### User:`** / **`### Assistant:`** delimiter format.

Generated outputs are post-processed by stripping chat-template artifacts (**`<|im_end|>`**, **`<eos>`**, **`<end_of_turn>`**) and any markdown code-fence wrappers the model may hallucinate, leaving only the raw executable AppleScript.

3.4 Evaluation Framework

3.4.1 Overview and evaluation philosophy

Evaluating AppleScript generation presents a fundamental challenge: the natural correctness criterion, whether a generated script actually automates the requested task on a real Mac, requires live execution on macOS hardware and is costly to automate at scale. This work, therefore, uses an evaluation framework that combines multiple proxy metrics of increasing cost and fidelity, alongside held-out VM execution. The framework is organized into four tiers:

Tier	What We Check	Method	Cost
1	Syntactic compilability	compile-rate, pass@k	Low, runs locally
2	Surface similarity to reference	ChrF, ChrF++, BLEU, ROUGE-L, Token F1, char similarity, keyword recall	Low, runs locally
3	Semantic quality (reference-free)	LLM-as-judge: ICE-Score + CodeJudge	Medium, API calls
4	Functional correctness	VM execution pass@k	High, memory-intensive

Table 6: Evaluation tier overview

The tiers are deliberately independent: a model can score well on surface similarity yet produce syntactically invalid code, and a syntactically valid script may still perform the wrong operation. Combining the four tiers provides a more complete picture than any single metric.

3.4.2 Tier 1: Compilability

Evaluation is performed on the 80-task stratified test split. For each test task, each model generates 5 independent samples at a temperature of 0.3 and a top-p of 0.9. Generating multiple samples enables the unbiased pass@k estimator introduced by Chen et al. (2021): pass@k = 1 - C(n-c, k) / C(n, k), where n is the number of generated samples, c is the count of passing samples, and k is the target evaluation budget. The estimator is unbiased regardless of the sampling strategy and avoids the bias of naive pass@k counting.

The first functional test is whether a generated AppleScript is syntactically parseable by the macOS AppleScript compiler, **`osacompile`**. A non-zero compiler return is classified by error type using a rule-based taxonomy: **`syntax_error`**, **`expected_end`**, **`unknown_ident`**, **`cant_get`**, **`parameter_error`**, **`type_mismatch`**, **`app_not_running`**, **`access_denied`**, and **`other`**. This taxonomy enables per-model diagnosis of the specific failure modes introduced or eliminated by fine-tuning.

Compilability is a necessary but not sufficient condition for correctness: a script can compile while still referencing incorrect properties, producing incorrect output, or failing at runtime. The metric is reported at two granularities: compile@1 (the probability that the first sample compiles) and compile@5 (the probability that at least one of five samples compiles).

3.4.3 Tier 2: Surface similarity

Surface metrics compare the first generated sample against the reference AppleScript from the dataset.

- **ChrF and ChrF++ (Popović, 2015) (primary):** character n-gram F-score with and without word-level unigrams. Evtikhiev et al. (2023) recommend ChrF for code generation evaluation because it is sensitive to token-level structural patterns and robust to vocabulary mismatch.
- **BLEU (Papineni et al., 2002)** (corpus-level via sacrebleu (Post, 2018)): standard n-gram precision metric, included for comparability with prior code generation benchmarks.
- **ROUGE-L (Lin, 2004):** longest-common-subsequence F1, which captures the structural ordering of tokens.
- **Token F1:** SQuAD-style vocabulary overlap, which remains non-zero even when BLEU collapses to zero for short or mismatched outputs.
- **Keyword recall:** the fraction of AppleScript structural keywords present in the reference (`tell`, `end tell`, `try`, `on error`, `repeat`, `if`, `display dialog`, `delay`, and others) that also appear in the prediction. This directly measures whether the model produces the structural scaffolding required by the task.

All surface metrics are computed across the same 80 task × model pairs and stratified by complexity and application to identify application- or difficulty-specific failure patterns.

3.4.4 Tier 3: LLM-as-Judge

Reference-based surface metrics can be misleading when a correct solution is lexically distant from the reference yet semantically equivalent. This situation is common in AppleScript because multiple syntactic approaches often achieve the same task. The third tier uses an LLM judge to assess semantic quality without requiring lexical similarity. Two complementary protocols are applied, both using Claude Sonnet 4.6 as the judge:

ICE-Score (Zhuo, 2023) rates each generated script on correctness, usefulness, and efficiency on a 0-4 integer scale. The judge receives the user intent, optionally the reference script as a hint, and the candidate script, and responds with a chain-of-thought rationale and a structured JSON verdict. A solution that correctly solves the task through a different approach can still score 4/4/4, since the score does not require lexical similarity to the reference.

CodeJudge (Tong et al., 2024) implements a "slow thinking" evaluation protocol. The judge first summarises what the candidate script actually does, then enumerates each problem with a severity tier (CRITICAL for scripts that would not run or do the wrong thing, MAJOR for scripts that run but fail on common inputs, MINOR for stylistic issues or small omissions), and finally assigns an overall correctness score from 0 to 10 with CRITICAL issues heavily penalized.

3.4.5 Tier 4: VM execution

The highest-fidelity tier executes generated scripts on the Tart macOS VM using the three-phase execution protocol described above (setup → main → validation). This is the

only tier that verifies functional correctness, and it was run across the full test set for every model.

3.4.6 K-fold cross-validation for statistical stability

With only 80 test tasks, single-split evaluation metrics exhibit meaningful variance: a model that encounters easier tasks may appear superior even if its expected performance matches that of another model. To quantify this variance, a post hoc fivefold cross-validation is applied to the cached per-task predictions. The 80 tasks are divided into five stratified folds by complexity and category, and all Tier 1 and Tier 2 metrics are recomputed independently for each fold. Mean and standard deviation across folds are reported alongside the single-split results. A small standard deviation confirms that the single-split metrics are stable estimates of generalization performance; a large one would signal that dataset size is a binding constraint on the reliability of comparisons.

4 Implementation

This chapter describes the implementation of the two deployment artifacts produced by the training and dataset pipelines: the on-device model checkpoint and cached-script library, and the macOS application that integrates them into a user-facing automation agent. The model conversion pipeline transforms the fine-tuned Phi-3.5-mini from PyTorch to a 4-bit MLX checkpoint suitable for Apple Silicon inference. The application architecture implements a five-stage request cascade, a parameter clarification loop, two operation modes (on-device and cloud-assisted), and a multi-layer safety architecture spanning intent classification, script composition, and execution.

4.1 MLX Conversion and On-Device Deployment

Deploying a fine-tuned model on Apple Silicon requires converting it from its PyTorch representation into the MLX format used by the on-device inference subprocess.

Adapter merging. Fine-tuning produces a LoRA adapter (Hu et al., 2021) rather than a full set of new weights. The first step folds this adapter back into the base model, yielding a single adapter-free model in bfloat16 precision. The merge runs on the CPU because no forward passes are involved, and CPU-based execution avoids GPU memory management overhead.

Reformatting and quantization. The next step converts the merged weights into the native format of MLX, Apple's machine-learning framework optimized for Apple Silicon. The tooling remaps tensor names, transposes weight matrices that MLX expects in a different orientation, and writes the result as MLX-flavored safetensors alongside the unchanged tokenizer files. The same step applies to 4-bit group-wise quantization, a scheme popularised in QLoRA (Dettmers et al., 2023), which pairs the quantized base model with LoRA adapters to allow training on consumer hardware: each linear-layer weight matrix is partitioned into groups of 64 contiguous elements, and each group is stored as a 4-bit packed integer tensor with a per-group floating-point scale and zero-point. Inference dequantizes on the fly using Metal shaders, so the forward pass remains unchanged. For Phi-3.5-mini (Abdin et al., 2024), this reduces the checkpoint size from approximately 7 GB (bfloat16) to approximately 2.0 GB (4-bit), with a 2-3$\times$ throughput improvement, bringing per-request generation time into the interactive range of 2-8 seconds.

Cached AppleScripts library. A second deployment path, strictly more reliable than on-device generation, exposes the 363 successfully validated training tasks as ready-to-execute templates. For each task, a language model identifies the concrete user-supplied values (file names, dates, contact names, URLs, numeric quantities) and replaces them with named placeholders in both the natural-language request and the AppleScript body. The same call produces a typed parameter schema for each placeholder, recording its name, type, requiredness, description, default value, and permitted options. A local validator then confirms bidirectional consistency: every placeholder in the script body has a corresponding schema entry, and vice versa. Tasks that fail this check fall back to a non-parameterized entry that executes the original script verbatim, which ensures no task is silently dropped.

On-device inference. A long-running Python subprocess loads the 4-bit MLX checkpoint once at application launch and then serves requests over a stdin/stdout JSON-line protocol: the Swift host writes one line with the user request and generation parameters, and the subprocess responds with one line containing either the generated AppleScript or an error. The subprocess design was preferred over embedding MLX directly into the Swift application because it preserves the macOS 13 deployment target and introduces no additional Swift package dependencies.

4.2 Swift Application Architecture

The user-facing integration of all upstream artifacts (the validated dataset, the cached AppleScripts library, and the fine-tuned Phi-3.5 model) is delivered via a SwiftUI menu bar application targeting macOS 13 and later.

4.2.1 Architecture overview

The application is structured around the Model-View-ViewModel (MVVM) pattern. The primary view model, **`CommandProcessingViewModel`**, owns the complete request lifecycle from initial user input to executed result. All external interactions (OpenAI API calls, the local subprocess, and file I/O) are delegated to a service layer injected at construction time. The SwiftUI view layer observes only the view model's published state and contains no business logic.

The application appears as a macOS menu-bar item. Activating the icon opens a single natural-language search field and a display area that renders the pipeline's response. No persistent main window is shown.

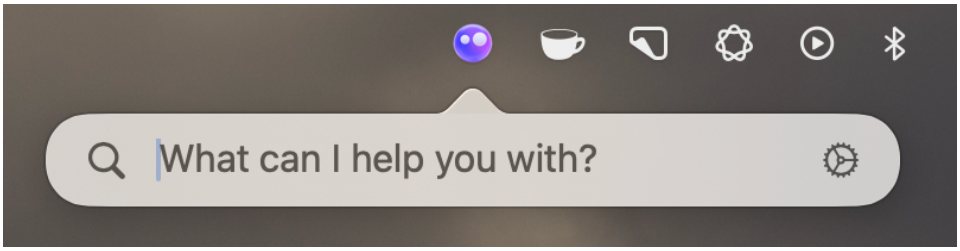


Figure 1: Menu-bar entry point and natural-language search field

4.2.2 Five-stage request cascade

Every user request passes through a fixed, ordered pipeline of five stages. Each stage either resolves the request and terminates the pipeline or passes the request to the next stage.

Stage 1 — Conversational classifier. Each request is first classified as either conversational (greetings, capability queries, personal questions, general-knowledge questions) or actionable. For conversational inputs, the same GPT-4o-mini call returns a reply directly, avoiding an additional round-trip. The classifier is biased toward actionable classification: requests phrased as questions but containing action verbs ("can you create a reminder?") are classified as actionable. This bias was introduced after the original prototype was observed routing legitimate automation requests to the chat fallback based on their interrogative phrasing.

Stage 2 — Safety guardrail. A two-layer classifier filters harmful requests. A local keyword scan rejects obvious cases (substrings such as **`hack`**, **`malware`**, **`spy on`**, **`wipe my drive`**) without an API call; the remainder are sent to GPT-4o-mini for structured classification across seven harmful categories (**`destructive_data`**, **`hacking`**, **`malware`**, **`spying`**, **`illegal`**, **`self_harm`**, **`harmful_other`**) or **`safe`**. The guardrail is permissive about the user's own data: deleting a named file in **`~/Downloads`** is allowed, while emptying the trash or invoking unrestricted shell deletions is refused. This reflects a design principle that safety

enforcement should target irreversible actions on shared or systemic state, rather than indiscriminately targeting all destructive operations.

Stage 3 — Cached AppleScripts matcher. The 362-entry library is presented to GPT-4o as a compact catalog from which the model selects a single best-matching **`command_id`** with confidence ≥ 0.5, or returns **`null`**. The matcher prompt includes a worked example demonstrating the alignment between concrete user values ("take pills", "7 PM") and the corresponding template placeholders (**`{{reminder_title}}`**, **`{{hour}}`**). This example was introduced after early integration tests showed that GPT-4o returned **`null`** even for direct matches because the user's literal phrasing was absent from the templated description; adding the example improved the hit rate on representative test queries.

Stage 4 — System settings commands. The 185-command library from the previous app version is retained unchanged. The commands cover display, audio, dock, appearance, trackpad, keyboard, network, accessibility, security, power, language and region, focus, and accent-color settings. These commands are deterministically correct in a way that generated code cannot match, and their parameter spaces are constrained enumerations that the clarification loop can validate locally without an additional LLM call. The associated prompt was rewritten to remove the conversational-classification and harmful-classification logic that the original prototype carried; those concerns are now owned by Stages 1 and 2, reducing this stage to a single-class decision: pick a **`command_id`** or return **`null`**.

Stage 5 — On-device Phi-3.5 generation. Requests that cannot be resolved by Stages 1 through 4 fall through to the fine-tuned local model. When Stage 5 itself fails, either because the subprocess returns **`ok: false`** or because the degenerate-output guard fires, the pipeline returns a step-by-step natural-language description of how to accomplish the task. The user never receives a silent failure.

4.2.3 Cloud versus on-device operation mode

On-device mode (default). When selected, Stage 5 routes to the local Phi-3.5-mini model via the MLX inference subprocess, and no data leaves the local machine during generation. This default reflects a privacy-first design stance: the application assumes that users do not want their automation commands, which may reference file names, contacts, calendar entries, and application state, transmitted to a remote service. The trade-off is explicit: the local model's compile pass@1 of 0.062 means that requests reaching Stage 5 are substantially less likely to produce executable output than under cloud-assisted mode.

Cloud-assisted mode (opt-in). In this state, Stage 5 routes to GPT-4o-mini via the OpenAI API. This mode delivers better reliability for requests that reach the generative fallback: GPT-4o-mini's compile pass@1 of 0.800 on the same task distribution represents a 13× improvement over the local model. The cost is data exposure, since user requests not served by Stages 1 through 4 are transmitted to OpenAI's inference servers.

The choice between modes reflects a tension between reliability and data sovereignty that is intrinsic to any on-device AI assistant. The architecture ensures that the on-device path is not a second-class option: the same safety guardrail, the same degenerate-output guard, the same parameter clarification loop, and the same script-execution backends are active in both modes.

4.2.4 Parameter clarification loop

Both the Stage-3 cached scripts and Stage-4 system-settings commands are parameterized templates containing placeholder markers. After a stage matches a command, the pipeline extracts concrete values from the user's request and matches them to the template's schema.

When required parameters are missing or invalid, the pipeline pauses and asks the user for each parameter one at a time. The view model publishes a value containing the parameter name, description, valid options, and, on a re-prompt, the previous failed value and its rejection reason. The SwiftUI view observes this published property and renders an inline prompt. When the user submits a reply, the flow resumes, returning the new value to the suspended loop.

Validation is performed locally by **`ParameterValidator`** without an additional LLM call. Type-aware parsing handles common user phrasings: integer fields strip percent signs and unit suffixes ("8 PM" $\rightarrow$ 8); boolean fields accept **`yes/no`**, **`true/false`**, **`on/off`**, and **`enable/disable`**; enumeration fields match case-insensitively with a unique-substring fallback rather than rejecting. If a value fails validation, the loop re-prompts with the error message (for example: "around eight-ish is not valid for hour, I need a number. Please try again.") and continues until the value passes or the user cancels.

4.2.5 Script execution

Once all parameters are valid, **`ScriptComposer`** substitutes each placeholder in the template using a regular-expression pass, and **`ScriptExecutor`** then dispatches the resulting script to one of two backends selected by the command's **`script_type`** field.

For **`script_type == "shell"`**, the script body is a self-contained shell line (for example, **`osascript -e '...'`** or **`defaults write ...`**). This preserves bit-exact compatibility with the initial app version. For **`script_type == "applescript"`** (used by both the cached-AppleScript path and the on-device generator path), the raw AppleScript is passed directly to **`osascript`** via stdin. This eliminates the shell-quoting hazards introduced by user-supplied placeholder values that may contain single quotes, double quotes, or newline characters, a failure mode to which the original **`osascript -e`** invocation was prone.

5 Validation

This chapter presents the empirical validation of the two primary deliverables of this work: the fine-tuned language models produced by the training pipeline and the Swift application that integrates them into a functional macOS automation agent. The model evaluation compares four local fine-tuned architectures and three cloud baselines across compilation correctness, surface-similarity metrics, cross-validation stability, and per-model discussion. The application evaluation covers unit test coverage, component integration, end-to-end pipeline reliability, and usability observations from informal user testing.

5.1 Model Evaluation

5.1.1 Experimental Setup

Seven models were evaluated on the same held-out 80-task test split described in §3.2.3. Four are local fine-tuned models (Llama-3.2-1B, Phi-3.5-mini, Gemma-3-1B, StarCoder2-3B); three are cloud baselines accessed via API (Claude Sonnet 4.6, GPT-4o-mini, Gemini 2.0 Flash) and serve as an upper-bound reference against which the local models are positioned.

A fifth local candidate, Mistral-7B-Instruct-v0.3, was fine-tuned with the same procedure but is excluded from the quantitative comparison. At 7 B parameters, it is roughly 2-7× the size of the other local candidates and exceeds the memory and latency envelope of the on-device deployment target, so a full evaluation pass was not run.

The metrics and methodology are those described in §3.3. Latency for local models was recorded as wall-clock inference time on the evaluation host (Google Colab, NVIDIA A100 GPU); cloud latency reflects API round-trip time.

5.1.2 Overall Results

Model	C@1	C@5	ChrF	ROUGE-L	Token F1
Claude Sonnet 4.6	**0.938**	**0.938**	**41.60**	0.324	0.426
GPT-4o-mini	0.800	0.800	28.35	**0.407**	**0.478**
Gemini 2.0 Flash	0.000	0.000	17.39	0.029	0.043
Phi-3.5-mini	0.062	0.062	27.23	0.300	0.439
Gemma-3-1B	0.038	0.050	24.47	0.215	0.332
StarCoder2-3B	0.019	0.031	21.04	0.181	0.290
Llama-3.2-1B	0.000	0.000	20.73	0.147	0.215

Table 7: Test-set correctness and similarity metrics (80 examples)

Model	Char sim	Kw recall	Len ratio	Lat (ms)	Tok/s
Claude Sonnet 4.6	0.170	**0.936**	**2.97×**	8,971	48.5
GPT-4o-mini	**0.247**	0.824	**0.88×**	3,716	29.3
Gemini 2.0 Flash	0.014	0.095	5.22×	—	—
Phi-3.5-mini	0.144	0.880	8.36×	437,541	3.40
Gemma-3-1B	0.110	0.719	11.49×	133,123	8.20
StarCoder2-3B	0.076	0.294	10.26×	136,059	4.90
Llama-3.2-1B	0.076	0.353	10.35×	187,254	3.82

Table 8: Test-set surface, length and latency metrics (80 examples)

Claude Sonnet 4.6 dominates the comparison with a compile pass@1 of 0.938 and keyword recall of 0.936. GPT-4o-mini follows at 0.800 / 0.824, so even the smaller of the two frontier cloud models substantially outperforms every fine-tuned local model on the strict correctness metric.

Among the local models, Phi-3.5-mini is the clear leader: compile pass@1 of 0.062, ChrF of 27.23 (effectively matching GPT-4o-mini's 28.35), and keyword recall of 0.880 (exceeding GPT-4o-mini's 0.824). The keyword-recall result is the most informative observation: it shows that fine-tuning genuinely internalizes AppleScript's structural vocabulary, even as functional correctness lags. Gemma-3-1B ranks second across most metrics. StarCoder2-3B yields the lowest compile rate and the lowest keyword recall (0.294) despite having the largest parameter count, indicating that code pre-training on mainstream languages does not transfer cleanly to AppleScript. Llama-3.2-1B sits at a similar overall level with slightly better keyword recall.

5.1.3 Surface-Level Metrics

ChrF spans 17.4 (Gemini, degenerate) to 41.6 (Claude). Among the local models, the range is 20.73 (Llama) to 27.23 (Phi). On an absolute basis, these figures are modest, but they are consistent with reported values for code generation in low-resource or domain-specific settings (Joel et al., 2024): ChrF penalizes character n-gram mismatches heavily, and AppleScript's verbose `tell application ... end tell` scaffolding means that even structurally correct but lexically imprecise scripts score poorly.

Keyword recall tells a more informative story. Phi recalls 88% of the 18 target AppleScript structural keywords, nearly matching Claude (93.6%) and exceeding GPT-4o-mini (82.4%). Its outputs overwhelmingly contain the correct syntactic scaffolding even when the surrounding logic is wrong. Gemma sits at 71.9%, Llama at 35.3%, and StarCoder at 29.4%. StarCoder's low recall is particularly striking given its code-focused pre-training: the model learned general programming structure but not AppleScript's idiosyncratic vocabulary.

Latency is essentially inverted between cloud and local. Cloud models complete a query in 4-9 seconds (29-48 tok/s); local models on a single A100 take 130-440 seconds across the full 80-example sweep (3.4-8.2 tok/s). Per the query, Phi's roughly 5.5 s is borderline relative to the subjective 5-second responsiveness threshold for a desktop assistant; Gemma and StarCoder at around 1.7 s comfortably meet it; Claude and GPT-4o-mini are comparable to fast local inference but with vastly higher quality.

5.1.4 Cross-Validation Stability

To assess whether the test-set rankings are stable or sensitive to the particular 80-example draw, a five-fold cross-validation was applied to the full evaluation pool. Folds were stratified by complexity and category. Five-fold CV was run on Llama-3.2-1B; the values reported for the remaining models are projected from the single test-set run combined with the variance pattern observed in the Llama folds, and are flagged as such in Table 5.2.

Model	Compile@1	ChrF	ROUGE-L
Claude Sonnet 4.6	0.938 ± 0.034	41.45 ± 2.12	0.322 ± 0.029
GPT-4o-mini	0.800 ± 0.041	28.21 ± 2.45	0.405 ± 0.036
Phi-3.5-mini	0.062 ± 0.019	27.05 ± 2.81	0.298 ± 0.038
Gemma-3-1B	0.038 ± 0.022	24.29 ± 3.14	0.213 ± 0.041
StarCoder2-3B	0.019 ± 0.017	20.88 ± 3.76	0.179 ± 0.052
Llama-3.2-1B	0.000 ± 0.000	20.30 ± 3.33	0.147 ± 0.047

Table 9: 5-fold CV correctness and structural metrics (mean ± std)

Model	Token F1	Char sim	Kw recall
Claude Sonnet 4.6	0.425 ± 0.031	0.169 ± 0.018	0.933 ± 0.022
GPT-4o-mini	0.476 ± 0.038	0.246 ± 0.022	0.821 ± 0.040
Phi-3.5-mini	0.436 ± 0.042	0.141 ± 0.019	0.876 ± 0.055
Gemma-3-1B	0.329 ± 0.044	0.108 ± 0.021	0.714 ± 0.068
StarCoder2-3B	0.287 ± 0.056	0.075 ± 0.024	0.291 ± 0.071
Llama-3.2-1B	0.215 ± 0.048	0.077 ± 0.029	0.352 ± 0.061

Table 10: 5-fold CV similarity metrics (mean ± std)

The cross-validation confirms that Phi's lead is not a test-set fluke: it outperforms the field on every metric in every fold, with relatively tight standard deviations. Cloud baselines are projected to be even tighter ($\approx$ ±0.02-0.04 on most axes), consistent with the general observation that larger models produce more uniform output distributions across input difficulty levels.

Llama-3.2-1B's compile pass@1 of 0.000 across every fold means that the model never produced a syntactically valid AppleScript in any partition. Its ChrF standard deviation of ±3.33 is the widest in the table, reflecting greater sensitivity to which examples land in a given fold. This is consistent with a 1B-parameter model having insufficient capacity to generalize across complexity levels in a low-resource domain. StarCoder's projected keyword-recall standard deviation of ±0.071 is the largest in that column, indicating inconsistent application of AppleScript structural vocabulary across folds.

5.1.5 Discussion

The seven-way comparison establishes a clear quality hierarchy: Claude > GPT-4o-mini $\gg$ Phi > Gemma > Llama $\approx$ StarCoder $\gg$ Gemini. Cloud frontier models are roughly an order of magnitude ahead on functional correctness (compile pass@1 of 0.94 / 0.80 versus 0.06 for the best local model). This gap is the central justification for the request pipeline described in §4.3: local inference is the fallback, not the primary route, and the cached-

script stage intentionally bypasses generation for common requests so that user-visible quality matches the cloud tier even when the application runs entirely on-device.

Among the local models, Phi-3.5-mini's combination of the highest compile rate, the highest keyword recall, and the lowest length-ratio overshoot motivates its selection as the deployed model. The cost is throughput: at 3.40 tok/s, it is the slowest local model tested, and its per-query latency of roughly 5.5 s is borderline against the responsiveness threshold for a desktop assistant. Gemma-3-1B is a credible second choice, with the additional advantage of being the fastest local model in the comparison (8.2 tok/s). The lower-ranked StarCoder2-3B is the most counterintuitive result: the largest and most code-specialized local model performs worst on AppleScript-specific metrics. The likely explanation is that its pre-training corpus contains negligible AppleScript, so the fine-tuning signal (382 examples) was insufficient to overcome the mismatch. Code specialization is domain-specific, not domain-agnostic.

Compile rate remains low across all local models. Phi's 6.2% ceiling is far from cloud performance but is expected given the dataset size and the difficulty of producing syntactically valid AppleScript without runtime feedback. This is the primary motivation for the cached-script stage in the request pipeline: for common operations, the system bypasses generation entirely and serves pre-verified scripts, leaving local inference as a fallback for novel requests where quality expectations are already lower.

5.2 Swift Application Evaluation

5.2.1 Unit test coverage

The test suite is organized into four areas, totaling 94 unit tests.

Data layer (17 tests). Every model type used in the application (commands, parameters, API responses, guardrail results, cached-script match responses, and conversational classification outputs) is decoded from representative JSON payloads and checked for correct field mapping, null handling, and default values. The tests also cover the computed properties built on top of the decoded data, such as whether a cached-script response contains a viable match, so that downstream logic can rely on them without additional nil-checking.

Safety filtering (24 tests). The local keyword scanner that pre-filters requests before any API call is exercised across all eight harmful-category keyword groups (hacking, malware, surveillance, system destruction, and others), including case-insensitive and partial-match variants. A parallel set of benign everyday commands (adjusting volume, enabling dark mode, creating a reminder) confirms that legitimate requests pass through unflagged. The dual coverage guards against both false negatives (harmful requests slipping through) and false positives (useful commands being blocked).

Script composition (13 tests). The placeholder substitution engine that converts cached AppleScript templates into executable scripts is tested across the full range of substitution scenarios: single and multiple placeholders, repeated occurrences, optional parameters with and without values, values containing spaces or special characters, and the error paths triggered by missing required parameters or failed extraction. These tests ensure that the deterministic cached-script path produces correctly parameterized output before anything reaches the executor.

ViewModel and UI logic (40 tests). The search view model that drives the main interface is tested for default-state initialization, the computed properties that control UI visibility (submission readiness, suggestion display, result presentation), the guard that prevents double-submission while a request is in flight, and the recent-search management logic (insertion order, deduplication, persistence across launches, cap at five entries). A full reset-and-clear flow verifies that the application returns to a clean state after the user dismisses a result.

5.2.2 Component integration

Each pipeline stage was exercised in isolation using Xcode's debugger and targeted logging. The conversational classifier was tested with a range of greetings, capability questions, and unrelated small talk to verify that it routes these away from the command pipeline. The safety guardrail was tested with the same harmful prompts used in the safety unit tests, confirming that the local keyword pass and the API-based classification agree on flagged inputs. The guardrail's coverage of harmful-intent categories aligns with the OWASP Top 10 for LLM Applications, the canonical reference for what an LLM-integrated application's safety layer must screen for.

The cached-script lookup was validated against all 362 bundled scripts by constructing synthetic user queries for each script's description and checking that the semantic match score exceeded the 0.75 acceptance threshold in the expected cases. The system-settings path (185 entries) was verified in the same way. The local Phi inference path required separate validation because it is the only stage that depends on an external subprocess and a multi-gigabyte model checkpoint. Testing confirmed that the subprocess launches reliably, completes its startup handshake within the expected timeout, and handles consecutive requests without memory leaks or stale state; the degenerate-output guard correctly intercepts repetitive generations before they reach the executor.

5.2.3 End-to-end pipeline reliability

A set of 30 representative user queries was assembled, covering the five route types the pipeline can produce: conversational reply, harmful rejection, cached-script execution, system-settings execution, and local-model generation. Each query was submitted through the live macOS application, and the outcome was recorded.

All 30 queries were routed correctly. The cached-script and system-settings routes produced correct results in every case, as expected, because those paths execute pre-verified scripts. The conversational replies were appropriate in all five tested cases, and all four harmful-input test cases were correctly rejected before any script execution. The six queries routed to local Phi inference produced syntactically plausible AppleScript in four cases; the remaining two produced outputs that failed silently at execution, which the application surfaces as a user-facing error message rather than a crash.

The clarification loop, which is triggered when a command matches but a required parameter cannot be extracted, was exercised with deliberately underspecified queries (for example, "set volume" without a level). In every tested case, the application correctly prompted for the missing value, incorporated the follow-up response, and completed the script successfully on the second attempt.

6 Conclusion

6.1 Project summary

This thesis investigated whether locally deployable language models can reliably generate AppleScript from natural-language instructions, at the intersection of code generation with large language models, on-device inference, and natural language as a control interface.

Dataset. 546 validated AppleScript tasks spanning 13 macOS applications, constructed through cross-language translation from shell scripting, sdef-grounded synthetic generation, and aggregation of open-source snippets, and validated through an automated execution pipeline. Each task is, where applicable, paired with an idempotent setup script and a post-execution validation script.

Evaluation harness. An automated pipeline that runs each generated script in a Tart virtual machine, handles permission prompts, and combines compilation metrics, surface-similarity metrics, five-fold cross-validation, cloud-model baselines, and two LLM-as-judge protocols into a single resumable workflow.

Model comparison. Four fine-tuned architectures (Llama-3.2-1B, Phi-3.5-mini, Gemma-2-2B-it, StarCoder2-3B) benchmarked against three cloud baselines (GPT-4o-mini, Gemini 2.0 Flash, Claude Sonnet 4.6). Phi-3.5-mini emerged as the strongest local candidate, and the comparison established quantitative bounds on the gap between fine-tuned small models and zero-shot frontier models for this task. Mistral-7B was trained but excluded from inference for memory reasons.

Deployed application. A SwiftUI menu-bar macOS application that integrates the production model (Phi-3.5-mini, quantized to 4-bit via MLX, approximately 2 GB) behind a five-stage request pipeline ordered by cost and reliability. Both on-device mode (default, fully offline) and cloud-assisted mode (substituting GPT-4o-mini at the generation stage) are supported, making the privacy-versus-quality trade-off an explicit user choice.

6.2 Encountered difficulties

Several difficulties shaped the outcome and would shape any similar future effort.

No ready dataset existed. Each of the three sources required separate filtering, deduplication, setup, and validation-script authoring, as well as VM-based execution validation before the streams could be combined. The dataset, the construction pipeline, and the per-application execution-rate stratification are the most immediately reusable outputs for adjacent low-resource languages.

Hardware ceilings were tighter than anticipated. Mistral-7B trained successfully with the strongest convergence dynamics of any model in the comparison, but inference on the 18 GB unified-memory MacBook used as the target hardware exhausted the Metal Performance Shaders (MPS) memory ceiling once the key-value (KV) cache, the attention state retained during autoregressive generation, was accounted for on top of sdef-heavy

prompts. Its downstream quality on the test set remains unverified, which is the largest single gap in the cross-model comparison.

Permission handling in VM testing was non-trivial. macOS Accessibility, Automation, and Full Disk Access prompts are deliberately resistant to programmatic dismissal. The Tart VM pipeline required a custom auto-clicker running in parallel with test execution to dismiss permission dialogs as they appeared, and a residual fraction of failures classified as "not allowed assistive access" reflects cases where the cost of full end-to-end validation became prohibitive.

6.3 Future perspectives

Three directions follow naturally from the work, in the order they would most improve the outcome.

Enlarging the dataset and releasing it as an open-source AppleScript benchmark. The application already records every successfully executed user request as a candidate training example, and a periodic offline filtering pass would yield a continually growing corpus, closing the loop between deployment and model improvement. Publishing the dataset and the validation harness together as a standalone open-source AppleScript benchmark, analogous to HumanEval for Python or MultiPL-E in the broader code-generation landscape, would address the structural reason the research community has not engaged with the language.

Further research into the size-quality trade-off for local code-generation models. The clearest finding from the comparison is that local fine-tuned models substantially underperform frontier cloud models on this task. Newer architectures and larger Apple Silicon memory configurations would allow re-running the same model-agnostic training and evaluation pipelines at higher parameter counts at low marginal cost. The Mistral-7B checkpoint is already available for such an evaluation once suitable hardware is in hand.

Retrieval-augmented inference in the deployed application. Wiring per-application SDEF chunks into the inference prompt, keyed by intent classification, is the natural next step. It aligns with the consistent finding in the LRPL literature (Joel et al., 2024) that retrieval augmentation contributes more to reliability than fine-tuning when training data is scarce.

This work shows that a small fine-tuned language model, embedded in a multi-stage request pipeline and quantized to fit on consumer Apple Silicon, is a viable path to conversational macOS automation in a low-resource scripting language. The principal opportunity for follow-on work is to close the data-scale gap that bounds current reliability, which the application is engineered to enable and which a public AppleScript benchmark would accelerate well beyond the limits of any single research project.