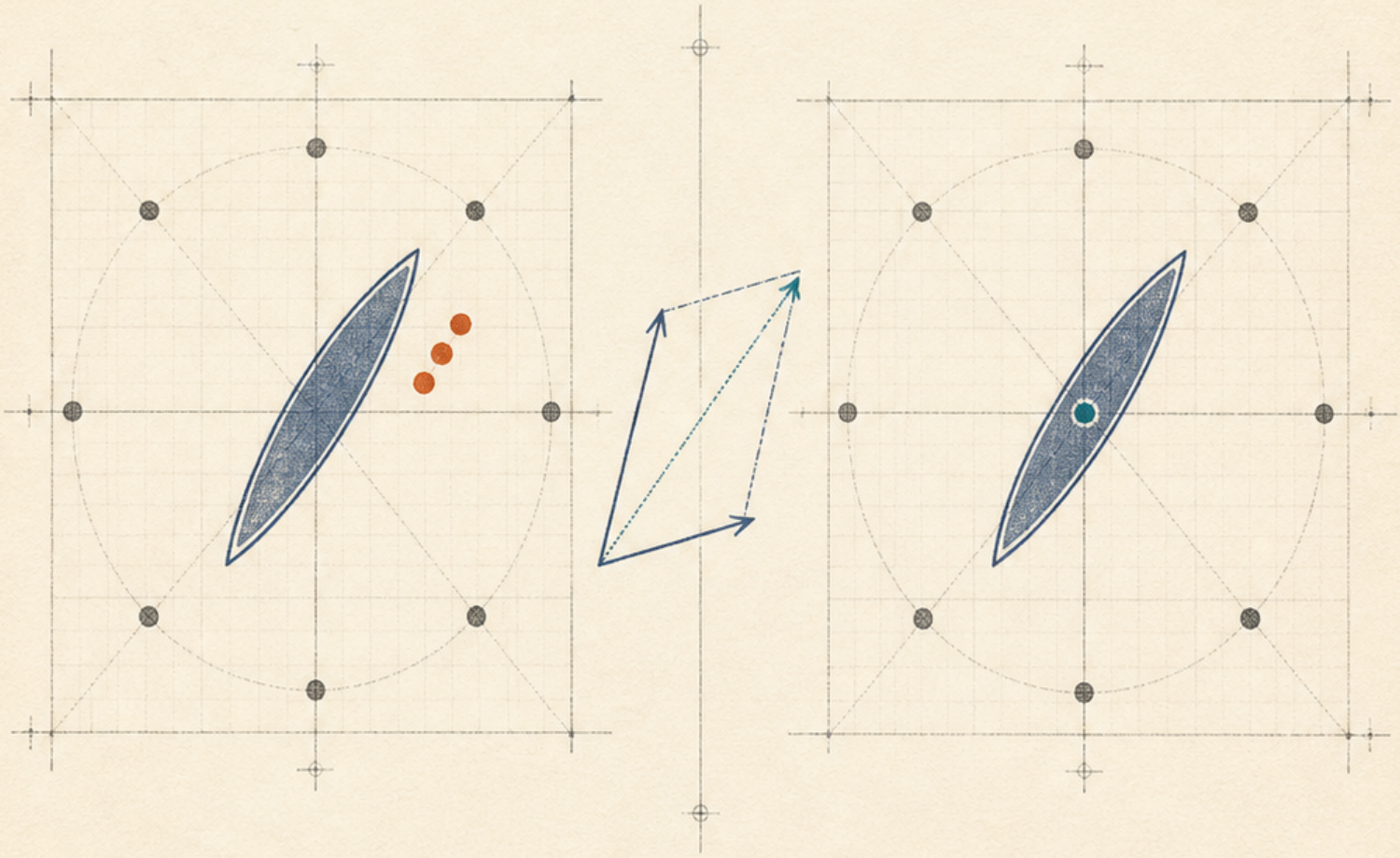
Mitigating API Hallucinations in Evolving Software Ecosystems with Controlled LoRA Fine-Tuning

Alexandru-Nicolae Ojica

Master of Science Thesis · Delft University of Technology



Mitigating API Hallucinations in Evolving Software Ecosystems with Controlled LoRA Fine-Tuning

by

Alexandru-Nicolae Ojica

in partial fulfilment of the requirements for the degree of

Master of Science
in Data Science and Artificial Intelligence Technology

at Delft University of Technology,
to be defended publicly on 27.08.2026.

Student number:	5477484
Project duration:	15.12.2025 – 30.08.2026
Thesis committee:	Prof. Dr. Arie van Deursen
	Prof. Dr. Maliheh Izadi
	Prof. Dr. Chirag Raman
	Daniele Cipollone
	Sergey Titov

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Abstract

Large Language Models (LLMs) generate fluent code that can still be invalid for the environment it must run in. Under fixed dependency versions a recurring failure is *API hallucination*: the model invokes a fabricated API, an outdated interface, or a plausible substitute for what the pinned library version requires. APIs released after the model's training data were collected are the hardest case, because the knowledge is missing outright and retraining the model for every library release is impractical.

This thesis asks whether small Low-Rank Adaptation (LoRA) adapters can inject post-cutoff API knowledge into a 7B instruction-tuned model (`Qwen2.5-7B-Instruct`) without corrupting its remaining behavior. An API counts as post-cutoff operationally when it entered the library in a recent release and the base model produces it on none of the tested surfaces. A teacher LLM generates training data grounded in library sources, checked mechanically and by manual review, and the adapters are trained with supervised fine-tuning (SFT) on target positives balanced with anchor and hard-negative examples that teach where the new API does not belong. The experiments cover one PyTorch API, then five, then twenty-five, all on the LibEvoBench completion benchmark.

A benchmark average cannot certify injection, so every claim is decomposed into per-API acquisition, target leakage onto unrelated code, a failure taxonomy separating fabricated, outdated, and wrong-but-valid answers, and retention on paired control benchmarks. Knowledge-free control adapters reproduce most of the aggregate benchmark gain with zero target output, and subtracting such a control from a trained adapter in weight space (task arithmetic) keeps target acquisition while moving unrelated rows at the measurement floor, so the acquisition metric, unlike the aggregate, tracks injected knowledge.

Twelve positive examples suffice to make the missing API appear, and the resulting adapter mentions it on 73.2% of unrelated rows, so the hard half of the problem is control rather than production. Anchored training restores that control, converting fabricated answers into correct ones without the anachronistic buildup of positive-only training. Both recipes then scale under replication across seeds. At five APIs the matched-control composition acquires 75.2% of the target rows while direct SFT converts only at seed-dependent peaks, and at twenty-five APIs, on the target rows the benchmark asks, the composition reaches 62.8% and direct SFT 50.1%, the direct recipe cutting the phantom share on those rows from 44.2% to 7.6%. The programme's one significant retention cost, a 2.88-percentage-point decline on GSM8K mathematical reasoning, belongs to the direct-SFT package, while both composition points change no paired control benchmark significantly, a contrast the cross-method design cannot attribute to a single component. The thesis therefore delivers a validated adaptation recipe together with the measurement discipline required to trust it, while release-scale coverage and a deployed library-version specialist remain open.

Preface and Acknowledgements

"Grown-ups never understand anything by themselves, and it is tiresome for children to be always and forever explaining things to them."

— Antoine de Saint-Exupéry

As researchers, we are constantly at risk of becoming these "grown-ups", of settling into comfortable assumptions and forgetting to ask why. To truly innovate, we must fight that urge and preserve the relentless, untamed curiosity of a child. We must approach the unknown not with the rigid certainty of adulthood, but with a humble, continuous desire to learn, to question, and to strive to be better every single day.

We spend much of our lives searching for absolute certainties, expecting the universe to hand us a predestined purpose or a definitive ground truth. But the universe does not offer explanations, it simply happens. It is a vast, indifferent canvas, and we are the ones tasked with forging our own meaning. I believe the true tragedy of growing up is forgetting that the unknown is not just a problem to be solved, but an adventure to be experienced. We innovate not because we are predestined to, but because we possess a duty to leave a piece of our soul in the things we build, pushing the horizon slightly further for those who will follow.

Yet, in a universe devoid of inherent purpose, the actual meaning we construct, the gravity that keeps us tethered to reality, is human connection. We create out of empathy, and we endure because of the people around us. I owe the courage to even step into this arena to my family. To my parents, who supported me through every year of university, making everything possible while constantly encouraging me to embrace new challenges. And to my brother, who pushed me from day one to simply be brave.

My journey has been shaped and sustained by three distinct forces. Thank you, Alex, for teaching me the true value of perseverance. In a landscape that constantly shifts, your unwavering consistency and steady presence have been my absolute foundation. Thank you, Ana, for being the most inspiring person I know. You possess a rare strength that doesn't merely endure the unknown, but steps into the future with a fierce, contagious enthusiasm. Your relentless ambition, bravery, and passion for innovation converge into a stubborn drive to change the world, constantly challenging me to dream beyond the horizon. And thank you, Vlad, a true visionary who taught me that life should never be taken too seriously, reminding me that a shared laugh is just as vital to the journey as fierce determination.

Meaning is also built through intellectual friction, and I have been incredibly fortunate in my mentors. I am deeply grateful to Sergey Titov for rigorously challenging my ideas every time we met, and to Daniele Cipollone for challenging me while supporting me every step of the way. I also want to thank Prof. Dr. Arie van Deursen for bringing a seasoned, high-level viewpoint to this work, offering the perspective necessary to navigate this terrain.

In artificial intelligence, we define a model's progress by minimizing a loss function against an absolute ground truth. In life, there is no universal ground truth, which means we carry the profound responsibility of defining our own loss function. I am deeply grateful to the people acknowledged here for helping me define mine.

Alexandru-Nicolae Ojica
Delft, August 2026

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
API	application programming interface
AST	abstract syntax tree
CDC	Critical Diff Check, a VersiCode migration metric
CPU	central processing unit
CUDA	Compute Unified Device Architecture, the NVIDIA GPU computing platform
DAG	documentation-augmented generation
DAPO	decoupled clip and dynamic sampling policy optimization
DARE	drop and rescale, a delta-sparsification operator
DPO	direct preference optimization
DS-1000	a data-science code-generation benchmark of 1,000 problems
EM	exact match
F1	F1 score, the harmonic mean of precision and recall
FIM	fill-in-the-middle
FRPO	fine-tuning robust policy optimization
GPT	generative pre-trained transformer
GPU	graphics processing unit
GRPO	group relative policy optimization
GSM8K	Grade School Math 8K, a mathematical-reasoning benchmark
IFEval	Instruction-Following Evaluation, a benchmark of verifiable instructions
ISM	identifier sequence match, a VersiCode completion metric
JIT	just-in-time (compilation)
JSON	JavaScript Object Notation
KTO	Kahneman-Tversky optimization, a preference-tuning objective
L0–L3	LibEvoBench context levels, from code only (L0) through redacted documentation (L1) and an added version requirement (L2) to the revealed API name (L3), defined in Chapter 2
LLM	large language model
LoRA	low-rank adaptation
MaHR	Macro Hallucination Rate, an APIHulBench metric
MBPP	Mostly Basic Programming Problems, a Python code benchmark
MiHN	Micro Hallucination Number, an APIHulBench metric
MMLU	Massive Multitask Language Understanding, a knowledge benchmark
MMLU-Pro	a harder multi-task extension of MMLU
MRR	mean reciprocal rank
ORPO	odds ratio preference optimization
PEFT	parameter-efficient fine-tuning
PM	prefix match, a VersiCode completion metric
pp	percentage points
PyPI	Python Package Index
RAG	retrieval-augmented generation
RL	reinforcement learning
RQ	research question
SEUS	Software Evolution Understanding Score, a LibEvoBench metric
SFT	supervised fine-tuning
SimPO	simple preference optimization
SVD	singular value decomposition

Abbreviation	Meaning
T1–T3	LibEvoBench task types (T1 masked-expression completion, T2 API identification from documentation, T3 signature production), defined in Chapter 2
TIES	trim, elect sign, and merge, a model-merging operator

1

Introduction

Large Language Models (LLMs) are neural language models built on the Transformer architecture and pretrained over broad text and code corpora [1], [2]. Their reach comes from adaptability: a single pretrained model can be prompted, shown a few examples, or lightly fine-tuned into a working system for translation, question answering, or programming, without training a separate model for every task [2], [3], [4].

Programming became one of the technology's main applications early. Codex showed that a language model trained on code can synthesize working Python from natural-language docstrings [5], and editor-integrated assistants have since made *code completion*, the task of continuing a partially written program or filling a marked gap in it, an everyday development tool that drafts functions, tests, and configuration from the surrounding project files [6]. Surveys of LLMs in software engineering report applications well beyond completion, across design, requirements, repair, refactoring, documentation, and analytics [6], but completion is where the model's output lands directly in a codebase, which makes it the setting of this thesis.

The weakness that shadows this usefulness is that the same interface produces fluent output whether or not anything supports it. A model prompted for something it does not know rarely abstains, because training and evaluation reward answering, so it guesses in the same confident register it uses when it knows [7], [8]. The literature calls such output *hallucination*, and this thesis uses the term for fluent or pragmatically plausible output that is unsupported, context-inconsistent, or factually wrong [9], [10], [11]. Every definition of this kind leans on an evidence source against which the output is checked, whether that source is verifiable world knowledge, the prompt and its context, or the model's own uncertainty [9], [11], and Chapter 2 follows that evidence source as it tightens from open-ended factual claims to code pinned to a particular library version (Table 2.3).

Code is the domain where that tightening ends in something checkable, because the evidence source stops being a matter of judgment: a generated fragment either parses, imports, runs, and passes its tests in a concrete environment or it does not. Code-hallucination studies accordingly organize the failures by the oracle that rejects them, from code that violates the stated requirements, through code that does not parse or run, to code that assumes packages, application programming interfaces (APIs), or files the project does not have [12], [13], [14]. Package hallucination illustrates the environment end of this spectrum, models recommending dependency names that look conventional but do not exist, often enough to open a package-confusion attack surface in the software supply chain [15], [16].

API Reliability in Evolving Libraries. This thesis narrows the preceding discussion to *API hallucination under fixed dependency constraints*, and it studies the failure in repository-aware Python code completion, completion conditioned on the files, imports, and configuration of a concrete project. That setting makes the failure measurable: a project's configuration pins its dependencies, each dependency is a library at one concrete version, and a completion that names an API the pinned version does not provide is invalid no matter how plausible it reads. Python supplies the ecosystem for this study, heavily used and fast-moving libraries together with public benchmarks that test the version-pinned setting directly, and within it PyTorch supplies the concrete APIs of every experiment. Against a pinned

environment, a completion can fail at the API level in four ways, and we call these the four *failure modes* of version-conditioned completion, under the following names:

- **Phantom APIs:** calls, attributes, imports, or classes that exist in no release of the target library.
- **Anachronistic APIs:** calls that exist in another release but not in the installed dependency version.
- **Wrong-but-valid substitutes:** APIs the pinned version does provide, used where a different API is required, often an older or semantically nearby alternative.
- **Signature mismatches:** valid API names used with invalid arguments, types, return assumptions, or call patterns.

The grouping is our own formulation, with established ingredients: API-oriented evaluation reports nonexistent calls and signature misuse as core failure categories and finds them concentrated where the model has seen the least, especially on low-frequency APIs, with documentation grounding improving reliability while staying sensitive to retrieval quality [17], [18], [19], and version-invalid usage is the failure that the version-aware benchmarks of Chapter 2 measure [20], [21]. Chapter 4 formalizes the first three modes, plus a no-API outcome for completions that yield no resolvable API, as the taxonomy that scores every experiment, classifying each prediction against the versioned public-API inventory built there, while signature validity is checked as a separate layer. The failure modes also interleave with ordinary task failure, since a model can miss the required operation altogether or understand it and still select an API the pinned version does not support, and the evaluation protocol of this thesis is built to keep those two failure sources apart.

One of these failure modes, the anachronistic call, exists only because libraries evolve: dependencies add, remove, rename, and change APIs between releases, so code that is correct for one release can be invalid for another, and the same evolution feeds wrong-but-valid substitution whenever the model prefers an API's older or better-known relatives. Several recent benchmarks measure completion under this version constraint [20], [21], [22], [23], [24], and the one the empirical chapters build on is LibEvoBench [25], a benchmark that takes completion tasks from real repository code, pins each task to a concrete library version, and scores whether the model produces the API that version requires. Its authors report two findings that motivate this thesis directly: models degrade on the APIs that change across versions while staying flat on the stable ones, and documentation in the prompt improves accuracy while merely stating the required version does not [25]. Chapter 2 describes this benchmark and its relatives in detail, and the introduction assumes no further familiarity with them.

Keeping a model's API behavior current as its libraries move is a maintenance problem of its own, one CodeSync frames as synchronizing a model's code knowledge with its evolving libraries [24], and this thesis calls it the *model update* problem. The *base model* in this framing is the model as its vendor shipped it, the pretrained and instruction-tuned weights whose knowledge is frozen at the time its training data was collected, and a model update is any change applied to that model after release so that its behavior tracks a world that has moved on, here the API surface of a library's newest releases. The concrete base model of this thesis is `Qwen2.5-7B-Instruct` [26], chosen because its weights are open and therefore trainable, because its instruction-tuned chat interface matches the completion protocol the benchmark uses, and because its seven billion parameters train on local hardware, with the full training and evaluation setup recorded in Chapter 5. Fully retraining such a model would keep it current but is far too costly to repeat for every release of every dependency, so the practical question is what smaller intervention can carry new API knowledge into a frozen base model. This thesis adopts the model-update framing and studies one such intervention.

The cost of leaving a base model stale follows from the object being executable. Because the dependency environment of a repository is fixed, a fabricated symbol or an API from the wrong release breaks the build, fails the continuous-integration run, or, in the package case, creates an exploitable installation path, and it arrives wrapped in code fluent enough to pass a quick review [13], [15]. The failures also concentrate on low-frequency and recently introduced APIs, which have the least public usage for a model to learn from [18], [20], and the newest APIs of a freshly released library version are the extreme point of that scarcity.

That extreme point, an API introduced after the model's training data was collected, is the cleanest version of the problem and the one this thesis studies. Studying it requires knowing which APIs the model cannot have learned, and no direct disclosure provides that knowledge: neither the Qwen2.5

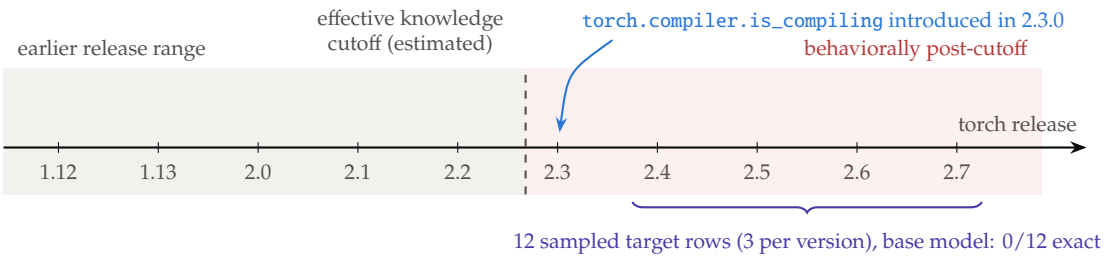


Figure 1.1: The operational post-cutoff setting studied in this thesis. The target API enters torch in version 2.3.0, and the sampled evaluation surface contains twelve target rows pinned to versions 2.4.0 through 2.7.0, on which the base model produces 0/12 exact target completions. The dashed marker is a schematic behavioral estimate, since Qwen2.5 has no disclosed calendar training-data cutoff.

technical report nor the `Qwen2.5-7B-Instruct` model card discloses a calendar training-data cutoff [26], [27]. Even a disclosed date would not settle the question, because a corpus collected on a date can still omit an older API or include a newer one, and temporal evaluation that trusts dates alone risks mistaking memorized prompt and solution patterns for changed knowledge [24], [28], [29]. We therefore use *post-cutoff* operationally, following LibEvolutionEval's behavioral estimation of effective knowledge cutoffs [20]: an API counts as *post-cutoff* when it entered the library in a recent release and the base model produces it on none of the tested baseline surfaces, the sets of benchmark requests that Chapter 5 defines. This zero baseline establishes behavioral absence on those surfaces, and it claims nothing about latent knowledge or corpus contents, so later successes measure acquisition against an observed zero rather than against an assumption about what the model once read. Figure 1.1 shows the construction for the concrete probe of the single-API experiments: the horizontal axis is the release history of `torch`, the dashed marker is the estimated effective cutoff (a behavioral estimate, since no calendar cutoff is disclosed), and the target API `torch.compiler.is_compiling` enters the library in version 2.3.0, on the far side of that estimate. The benchmark contains 30 completion rows whose correct answer is this API, of which the sampled broad surface of the single-API experiments holds twelve, three for each of the versions 2.4.0 through 2.7.0, and the base model completes none of the 30 with the target, the observed zero from which every later acquisition claim is measured.

A post-cutoff API defined this way turns the model-update problem into a measurable experiment, because the missing behavior is identified, the baseline is zero, and any later production of the target is attributable to the intervention. The intervention this thesis studies is controlled fine-tuning with Low-Rank Adaptation (LoRA), which trains a small additional set of weights, an *adapter*, while the base model stays frozen [30]. The adapter's training material is *source-grounded synthetic data*: synthetic because every training example is written by a teacher LLM, a separate stronger model used only as a data generator (Claude Sonnet 4.5 in this thesis [31]), and source-grounded because that teacher works from each target API's official sources, its documentation, signature, and tests. The question is whether this data and such an adapter can reduce post-cutoff API hallucinations while three risks stay controlled: the new API spreading onto code where it does not belong, damage to behavior that was already correct (a known risk of fine-tuning on new facts [32]), and apparent improvement that only reflects familiarity with the benchmark task.

The long-term motivation behind that question is *library-version specialization*, adapting a model so that its API behavior matches one concrete dependency environment, and Figure 1.2 shows the pipeline this goal implies. The pipeline starts from the target dependency setting, a library pinned at one version. Public-API extraction then builds a versioned inventory of the interfaces that version documents for its users, and this inventory serves twice, as the source from which target APIs are chosen and as the oracle against which any generated API name can be checked for existence and version validity. Synthetic example generation produces the source-grounded examples for the chosen targets, controlled LoRA fine-tuning trains the adapter on those examples, and API-level evaluation measures what the adapter established, what it spread onto unrelated code, and what it damaged. The experiments of this thesis implement the adaptation and evaluation stages of this pipeline as controlled probes and stop short of a complete specialist system.

A specialist adapter must eventually carry many APIs at once, and jointly taught APIs compete. Siblings from the same namespace contend for the same completion contexts, an example that teaches

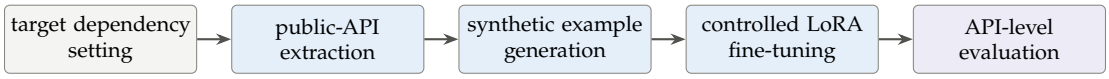


Figure 1.2: Pipeline structure behind the long-term goal of library-version specialization. The experiments implement the adaptation and evaluation stages as controlled post-cutoff API probes and stop short of a complete specialist system.

one API doubles as a context in which the others must stay silent, and the emission pressure that establishes one target can push a related target onto rows where it is wrong. Whether adaptation survives this *competition among targets* cannot be extrapolated from a single-API result, so the experimental programme is staged by scale.

Each scale answers a question the previous one cannot. The one-API stage isolates the mechanisms, because with a single target, acquisition, leakage, and retention separate cleanly, every training example and benchmark row can be inspected by hand, and a failure can only come from a design choice, since no interaction among targets exists yet. The five-API stage introduces the competition described above at a size where the full measurement protocol, per-API scores, leakage accounting, and replication across training seeds, still runs for every target. The twenty-five-API stage then asks the transfer question, freezing the recipes validated at five APIs and applying them, without per-API tuning, to twenty APIs they were never developed against, which tests whether the earlier conclusions describe the recipes or only the APIs they were tuned on. Together the three scales connect the controlled probe to the coverage a library-version specialist would need, with twenty-five APIs still an intermediate point on that path.

1.1. Problem Statement

This thesis addresses API hallucination in repository-aware Python code completion under fixed dependency constraints, where the target library and version are known. A completion fails at the API level when it produces a call, attribute, or signature usage that is invalid for the target environment or when it omits the required target API, and Chapter 4 formalizes these outcomes as the API-level taxonomy that scores every experiment.

We treat API hallucination reduction as a joint problem of acquisition, hallucination suppression, specificity, retention, and scalability, measured at one, five, and twenty-five APIs under the same benchmark family. The intervention is an API-specific LoRA adapter trained from source-grounded synthetic data whose validation status is recorded explicitly, and it serves as an experimental probe of whether missing API behavior can be established under measured acquisition, leakage, locality, and retention constraints. LibEvoBench is the measuring instrument throughout, and runtime discovery of the target API, through retrieval-augmented generation or otherwise, is outside the task.

Adapter routing and serving-time composition appear only in the discussion, because the task-vector method used in the experiments is a training-time construction of one adapter from paired arms rather than deployment-time routing among adapters. API identity and version validity are the primary evaluation objects, general program correctness enters only through the separate retention suite, and the twenty-five-API extension covers an intermediate scale while full coverage of a library release stays open.

1.2. Objective and Research Focus

The objective is to determine whether source-grounded synthetic data and controlled LoRA adaptation can reduce API hallucinations for post-cutoff APIs under fixed dependency constraints, and the one-API probe makes acquisition, specificity, and retention measurable before the five-API and twenty-five-API stress tests. Chapter 5 defines the exact evaluation surfaces derived from LibEvoBench.

We train on synthetic examples generated by a teacher LLM rather than on mined real-world code, because the intended setting is adaptation at day zero, when a new library version has just been released and the APIs of interest have few or no public usage examples to mine, while their documentation, signatures, and tests are already available. A teacher model working from these sources can produce training examples before the new API appears in real-world code, and it can produce them in controlled proportions for each role the training design needs: positives, examples that require the new API, anchors, examples whose correct answer is an already-known API, and hard negatives, close valid alternatives for contexts where the new API is wrong. Deliberate generation also reaches what real code

rarely shows, the negative space around an API, the contexts in which it must not be used. Provenance stays auditable because every example traces back to a named source artifact, contamination is controlled because nothing is copied from the evaluation benchmark, and every generated row can be mechanically validated against the versioned API surface at construction time. Chapter 4 develops these roles and the validation workflow in detail, and Chapter 6 reports which checks each corpus actually passed.

The primary objective measures the reduction through structured API-level outcomes instead of one aggregate score, using exact target API match, the taxonomy rates, target leakage, and movement on unrelated rows that Chapter 4 defines. A secondary objective identifies which data and training conditions establish conditional target use and which only calibrate the benchmark task, by comparing mixtures of the three data roles, task-diverse prompt formats, matched format controls, and adapter construction choices. The final objective asks whether the one-API conclusions survive target competition over five and then twenty-five APIs, where the recipes are applied unchanged, under criteria fixed before launch, to the twenty new APIs.

Five progressive research questions (RQs) organize these objectives, each producing the evidence the next one needs: RQ1 diagnoses how the base model fails on version-conditioned completion, RQ2 asks whether source-grounded synthetic data can encode the behavior those failures call for, RQ3 tests whether a small LoRA adapter can teach one post-cutoff API while reducing the hallucinations that replace it, RQ4 separates conditional target use from benchmark alignment, leakage, and interference, and RQ5 follows the recipes to five and then twenty-five APIs. Chapter 3 states the five questions formally, with the success reading each one carries.

1.3. Contributions

The work contributes the following artifacts:

- A formulation of post-cutoff API hallucination as a measurable subproblem of library-version specialization, resting on the operational zero-baseline definition above.
- A workflow that generates source-grounded synthetic examples from API signatures, documentation, tests, and availability metadata, renders them deterministically into training tasks, and records which validation checks each corpus passed.
- A controlled adaptation methodology that compares direct LoRA training with matched-control task-vector composition.
- An API-level evaluation protocol covering exact target use, the wrong-but-valid, phantom, anachronistic, and no-API outcome classes, target leakage, paired movement, and retention, together with an instrument audit that uncovered a coverage gap in the benchmark's documentation-bearing context levels and corrected the affected denominators and selection gate.
- Seed-replicated evidence at one, five, and twenty-five APIs, including the gap between contained and scored API knowledge, target competition, and the retention effects needed to interpret the successful operating points.

2

Background and Related Work

2.1. How Hallucinations Are Defined in the Literature

2.1.1. General LLM Hallucinations

General LLM studies typically define hallucination as plausible output that is non-factual or misaligned with input context or instructions [9], [10]. These definitions recur in a two-way split between factuality failures (knowledge mismatch) and faithfulness failures (instruction, context, or reasoning mismatch) [9]. Kalai et al. frame hallucinations as uncertainty-driven guessing amplified by training and evaluation incentives, where LLMs learn that giving an answer is more profitable than not giving an answer, especially in contexts where the model has very low confidence [7].

Benchmark work turns these definitions into measured tasks: HalluLens separates intrinsic from extrinsic hallucination with dynamically generated tests [11], and the Hallucinations Leaderboard evaluates factuality and faithfulness across several task families [10].

Mitigation studies report improvements under narrow conditions, through detection-and-correction pipelines and abstention mechanisms [8], [33], and Gekhman et al. show, on the training side, that fine-tuning on new facts can increase hallucination tendency as those facts are learned [32], the update risk that the retention axis of this thesis measures. Table 2.1 summarizes these definition lenses and the evidence each paper reports.

2.1.2. Code-Level Hallucinations

Where the general definitions above must choose an evidence source, code-hallucination papers inherit one from execution and define failure at the executable artifact level (Table 2.2). Liu et al. conduct a

Table 2.1: General LLM hallucination definitions and paper-reported evidence.

Work	Definition Lens	Taxonomy / Axes	Paper-Reported Signal
Huang et al. survey [9]	Hallucination as plausible but unsupported output	Factuality vs. faithfulness split and subtype taxonomy	Survey synthesis spanning causes, detection, and mitigation
Kalai et al. [7]	Hallucination as uncertainty-driven guessing	Statistical/incentive-misalignment framing	Explains persistence via benchmark scoring pressure
HalluLens and Hallucinations Leaderboard [10], [11]	Benchmark-centered operational definition	Intrinsic/extrinsic and factuality/faithfulness dimensions	Dynamic benchmark construction and multi-task model comparison
ABSTAIN and update-risk work [8], [32]	Overconfident forced generation and unstable updating	Single-token abstention mechanism and post-training risk analysis	95% $\rightarrow$ 0% in the controlled single-token setup, while update learning can raise hallucination tendency

Table 2.2: Code-level hallucination definitions with dataset scale and quantitative signals from the original papers.

Work	Primary Definition	Taxonomy / Categories	Scale / Reported Signal
Liu et al. [12]	Conflict with requirements, context, or knowledge	Three primary categories and 12 specific types	3,120 generated samples with prompt-based mitigation explored
CodeMirage [13]	Hallucination as plausible but defective generated code	Dead or unreachable code, syntactic incorrectness, logical error, robustness issue, security vulnerabilities	1,137 GPT-3.5 hallucinated snippets from HumanEval and MBPP
CodeHalu [14]	Execution-verified mismatch with required behavior	Mapping, naming, resource, logic	CodeHaluEval: 8,883 samples, 699 tasks, 17 LLMs
Spracklen et al. [15]	Dependency-level factual conflict	Package confusion and hallucinated dependencies	DeepSeek fine-tuning reaches 2.66% hallucination while pass@1 falls from 51.4% to 25.3%
Krishna et al. [16]	Package hallucination as supply-chain vulnerability surface	Cross-language/model risk-factor analysis	Hallucination rate varies by model, size, language, and prompt specificity

thematic analysis over 3,120 LLM-generated code samples and define three primary categories with 12 specific types before exploring prompt-based mitigation [12], while CodeMirage defines hallucinated code as plausible but defective, with five categories covering dead or unreachable code, syntax, logic, robustness, and security, and a benchmark of 1,137 GPT-3.5 hallucinated Python snippets from HumanEval and MBPP [13]. CodeHalu instead verifies failure by execution, with mapping, naming, resource, and logic classes, and evaluates 17 LLMs on 8,883 CodeHaluEval samples from 699 tasks [14].

Package-hallucination studies add a security lens: Spracklen et al. generate 576,000 samples across 16 models and find average hallucinated-package rates of at least 5.2% for commercial and 21.7% for open-source models, with 205,474 unique hallucinated names [15]. Their fine-tuning mitigation cuts DeepSeek's package-hallucination rate to 2.66% while its HumanEval pass@1 falls from 51.4% to 25.3%, a repaired hallucination bought with general coding ability. Importing Phantoms likewise finds variation by model, language, model size, and task specificity, together with an inverse relationship to HumanEval performance [16].

2.1.3. API Hallucinations in Version-Specific Code

API hallucinations narrow code hallucinations to interface validity under concrete dependencies, including non-existent API calls, signature misuse, unavailable imports, and version-mismatched usage [17], [18], [19]. AutoAPIEval defines API recommendation and API-oriented code-example generation with four metrics [17]. `IncorrectAPI` combines API nonexistence and complete-signature mismatch for recommendation, while `NoAPIInvoked`, `Uncompilable`, and `Unexecutable` measure three separate failures in generated examples.

CloudAPIBench counts an invocation as valid only when it binds successfully against a stub mirroring the API's documented signature, a check on the method name and the argument structure [18]. GPT-4o low-frequency validity rises from 38.58% to 47.94% with documentation-augmented generation, while a suboptimal retriever causes a 39.02-point drop on high-frequency APIs. Selective triggering is a separate comparison that raises overall GPT-4o validity from 66.40% to 74.60%, an 8.20-point gain [18]. MARIN/APIHulBench defines Micro Hallucination Number (MiHN), the mean number of hallucinated API elements per generated API, and Macro Hallucination Rate (MaHR), the share of generated APIs containing at least one hallucination [19]. The paper reports reductions of 67.52% in MiHN and 73.56% in MaHR relative to its retrieval-augmented generation (RAG) baseline, with separate results on internal industrial projects [19].

Version-aware evaluation then tests API hallucination under evolving libraries. LibEvolutionEval covers eight libraries through annual snapshots over multiple years, reporting API-call F1 for completion and mean reciprocal rank (MRR) for version-specific documentation retrieval [20], and VersiCode covers

Table 2.3: Cross-level matrix: how hallucination definitions become stricter from natural language to version-specific code.

Level	Evidence Source	Typical Failure Unit	Representative Metrics / Benchmarks
General LLM hallucination	World knowledge, prompt/context, model uncertainty [7], [9], [11]	Unsupported factual claim or unfaithful response behavior	Factuality/faithfulness benchmark suites, intrinsic/extrinsic tasks [10], [11], [34], [35]
Code-level hallucination	Program requirements, execution outcomes, dependency correctness [12], [13], [14]	Defective code artifact in the logic, syntax, resource, or dependency class	Liu taxonomy, CodeMirage, and CodeHaluEval category-specific evaluation [12], [13], [14]
API hallucination (version-specific)	Library docs, dependency graph, target package version [17], [18], [19]	Invalid recommendation, signature mismatch, version-incompatible call or import	AutoAPIEval rates, MiHN/MaHR, Pass@k, CDC@k, and executable tests [17], [19], [21], [22], [23], [24]

version-specific completion and version-aware code migration over more than 300 Python libraries and more than 2,000 versions, scored with Pass@k, identifier sequence match (ISM@k), and prefix match (PM@k) for completion and Critical Diff Check (CDC@1/3) for migration [21].

GitChameleon contains 116 executable version-conditioned tasks, on which GPT-4o solves 39.9% at pass@10, or 43.7% with error feedback, a task counting as solved when at least one of ten completions passes its tests [22], and GitChameleon 2.0 extends the executable evaluation to Python library incompatibilities with hidden-test success and API hit rate [23]. CodeSyncBench covers 220 APIs from six libraries with 3,300 test cases and 2,200 update-aware training samples, and its evaluation of 14 LLMs reports persistent synchronization errors even after direct preference optimization (DPO), odds ratio preference optimization (ORPO), and simple preference optimization (SimPO) updates [24].

LibEvoBench evaluates the same completion tasks across pinned library versions while separating evolving from stable APIs [25], and because the empirical chapters use it as their measuring instrument, Section 2.2.1 describes its tasks, context levels, and findings in full. This thesis defines API hallucination operationally as *version-invalid API behavior under repository-aware completion*.

To make the transition from language-level hallucinations to repository-level API failures explicit, Table 2.3 provides a cross-level definition matrix grounded in the cited papers.

2.2. Benchmarks for API Reliability Under Library Evolution

The strictest level of the matrix above, version-specific code, is where recent evaluation has moved: benchmarks now measure API validity jointly with executability and update synchronization in dynamic version-aware settings. Table 2.4 summarizes benchmark scopes and headline quantitative signals reported by the papers.

Across the benchmark families, low-frequency and dependency-constrained APIs produce more errors than well-represented APIs [18], [19], performance changes across library versions and years [20], [21], [22], [24], and update-aware post-training reduces but does not remove the errors [24]. These findings motivate measuring version-specific target behavior together with non-target movement and retention.

2.2.1. LibEvoBench

LibEvoBench, the measuring instrument of the empirical chapters, evaluates completion tasks mined from real public repository code across pinned library versions while separating evolving from stable APIs, a design its authors call *temporal knowledge stratification* [25]. It defines three task types, which the paper names API Calling, API Identification, and Signature Recall and which this thesis abbreviates as T1, T2, and T3: T1 masks one API expression in a real code context and asks for the replacement, T2 asks for the API name given its module path and a description with the name redacted, and T3 asks for the signature of a named API, scored by parameter-name overlap. Completion prompts render at four context levels, where L0 gives the code only, L1 adds a redacted docstring that describes the target API without naming it, L2 further adds an explicit version requirement of the form `# requires: lib==version`, and L3 replaces the documentation with the revealed API name next to the

Table 2.4: Comparison of benchmark resources used to study API reliability under evolving software ecosystems.

Benchmark	Scope Reported in Paper	Version-Aware?	Task / Metric Focus	Headline Signal from Paper
AutoAPIEval [17]	Two tasks and four metrics in a three-LLM case study	Partial	IncorrectAPI, NoAPIInvoked, Uncompilable, Unexecutable	Reports model variation across the two tasks
CloudAPIBench [18]	Frequency-annotated API benchmark	Implicit	Invocation validity and retrieval strategy	Selective triggering raises GPT-4o overall validity from 66.40% to 74.60%
APIHulBench (MARIN) [19]	Six-LLM benchmark with dependency-aware decoding constraints	Project-aware	API hallucination counts/rates (MiHN/MaHR)	Versus RAG, MiHN is 67.52% lower and MaHR is 73.56% lower
LibEvolutionEval [20]	Eight libraries across annual snapshots over multiple years	Yes	API-call F1 and documentation-retrieval MRR	Reports variation across library evolution
VersiCode [21]	>300 libraries and >2,000 versions across nine years	Yes	Completion with Pass@k, ISM@k, PM@k and migration with CDC@1/3	Reports difficulty with version control
GitChameleon [22]	116 executable version-conditioned problems	Yes	pass@k with executable tests	GPT-4o pass@10 is 39.9%, or 43.7% with feedback
GitChameleon 2.0 [23]	Follow-up benchmark focused on Python version incompatibilities	Yes	Hidden-test success rate and API hit-rate analysis	Extends executable evaluation of version-specific failures
CodeSyncBench [24]	220 APIs, six libraries, 3,300 tests, and 2,200 update-aware samples	Yes	Synchronization with real API updates	Reports persistent errors after preference updates
LibEvoBench [25]	Version-pinned completion on evolving and stable APIs	Yes	Temporal stratification and SEUS	Reports that documentation helps while version strings alone do not
CodeHaluEval [14]	8,883 samples over 699 tasks for execution-based hallucination analysis	Indirect	Mapping, naming, resource, and logic error profiling	Complements API benchmarks with broader code hallucination diagnostics

Table 2.5: Retrieval-oriented baselines in API hallucination papers and their reported trade-offs.

Work	Retrieval Baseline Form	Paper-Reported Benefit	Paper-Reported Limitation
CloudAPIBench [18]	Documentation-augmented generation (DAG) for API completion	Improved low-frequency API validity with DAG and selective triggering	Performance can degrade under sub-optimal retrieval/triggers, especially on high-frequency APIs
APIHulBench (MARIN) [19]	RAG baseline versus hierarchical dependency-aware mitigation	Lower MiHN and MaHR than RAG on benchmarked settings	RAG remains the benchmark comparator, and industrial-project results are reported separately

version requirement, shifting evaluation at that level to the predicted parameters. On this design the authors report that state-of-the-art models are largely version-oblivious, with performance degrading on evolving APIs while remaining flat on stable ones, and that the documentation added at L1 improves accuracy while the version requirement added at L2 does not [25]. Which task type and context level this thesis evaluates, and why the remaining ones are excluded, is fixed in Chapter 4.

2.2.2. Retrieval-Augmented Baselines

RAG provides an inference-time comparison because it supplies external API information without changing the base weights. CloudAPIBench reports gains from documentation-augmented generation together with sensitivity to retriever quality and triggering strategy [18]. MARIN/APIHulBench compares its hierarchical dependency constraints with RAG and reports lower MiHN and MaHR in the evaluated settings [19] (Table 2.5).

2.3. Temporal Generalization and Memorization Risks

Evaluation protocols for evolving APIs also need to separate true adaptation from memorization effects. EvoCodeBench argues that data leakage and missing domain-specific evaluation can bias code-generation benchmarking [28]. Memorize-or-Generalize reports that LLM code generation can depend heavily on memorized prompt and solution patterns and that performance drops on evolved

problem variants, indicating fragile generalization [29].

These findings matter for version-conditioned completion because overlap between adaptation examples and benchmark prompts can inflate apparent API acquisition. The methodology therefore records source provenance and checks complete prompt-answer overlap between the synthetic corpora and the benchmark before interpreting downstream utility.

2.4. Fine-Tuning Strategies for Reducing API Hallucinations

2.4.1. Update-Aware Supervision and Preference Optimization

The CodeSync result above, residual synchronization errors after update-aware instruction tuning and preference optimization, motivates objectives and controls that measure target behavior directly [24].

2.4.2. Reinforcement Learning for API Knowledge Updating

ReCode proposes rule-based reinforcement learning for outdated API behavior, using an approximately 2,000-example migration dataset and rewards based on code similarity under updated constraints, and reports stronger adaptation on unseen API-update tasks with less degradation of general coding ability than its supervised fine-tuning baseline [36]. Its optimizers include group relative policy optimization (GRPO), which DeepSeekMath established at scale in mathematical reasoning [37], and decoupled clip and dynamic sampling policy optimization (DAPO), a later refinement of GRPO [36], [38], so such pipelines are feasible even though none of this work supplies evidence about versioned APIs.

2.4.3. Catastrophic Forgetting and Hallucination Risk During Updating

The update risk of Gekhman et al., cited above, belongs to the training stage of the same problem, and fine-tuning robust policy optimization (FRPO) responds to it by optimizing policies for stability under later fine-tuning shifts, reporting reduced forgetting in its post-training experiments [32], [39]. API-update evaluation therefore needs both target acquisition and retention of previously reliable behavior.

2.5. LoRA-Based Version Specialization

For the per-release updating that library evolution demands, the cost of each update matters, and parameter-efficient fine-tuning (PEFT) addresses it by training a small set of additional parameters while the base weights stay frozen. **LoRA** is the reference instance of this idea for large language models: it represents the weight update of selected matrices as a product of two low-rank factors, which reduces the trainable parameters and optimizer state by orders of magnitude, and the learned update can either be merged into the base weights or kept as a small separate artifact [30]. Three consequences make this a plausible mechanism for library-version specialization: adaptation becomes cheap enough to repeat for every release, the per-version artifact is small enough to store and distribute in large numbers, and the base model remains shared, so adapters for many dependency environments can coexist around one deployment.

The LoRA literature has since branched along the axes that such a per-version workflow would stress, lowering the cost of the adaptation step [40], stabilizing and budgeting the low-rank update [41], constraining it against interference with pretrained knowledge [42], shrinking the per-adapter storage footprint [43], and serving many adapters concurrently over one base model [44]. None of this work, however, asks whether an adapted model uses a newly taught API only where that API is correct.

LoRA thus supplies a controlled adaptation substrate in which the base model remains frozen while a small update carries the target pressure, and Chapter 4 builds the experimental probe on that property, while per-version routing, production serving, and dynamic composition among deployed adapters remain outside the empirical scope set in Chapter 1.

2.6. Synthesis and Research Gap

Across the reviewed benchmarks, API and version errors persist at benchmark scale over evolving libraries and long time ranges [20], [21], [23], [24], [25], retrieval helps when the right documentation is found but stays sensitive to retrieval quality and triggering [18], [19], and training-side updates report gains together with residual errors or forgetting risk [15], [32], [36], [39]. The PEFT literature supplies an efficient substrate for controlled weight updates, yet nothing in it determines whether a benchmark gain

reflects use of the intended APIs in the right contexts, calibration to the benchmark task, or emission of newly taught symbols where they do not belong [30], [40], [41], [42], [43], [44].

No reviewed study, however, starts from an API the model demonstrably cannot produce, teaches it from synthetic examples grounded in the library's own sources, and then measures side by side whether the model uses the API where it is required, emits it elsewhere, and keeps its unrelated behavior intact as the number of taught APIs grows. This thesis studies that combination through controlled LoRA adaptation and task-vector composition, as formalized in the research questions of Chapter 3.

3

Research Questions

Chapter 2 closed with the gap that no reviewed study starts from an API the model demonstrably cannot produce, teaches it from synthetic examples grounded in the library's own sources, and measures target use, misuse, and unrelated behavior together as the number of taught APIs grows. Five research questions turn that gap into the experimental programme of this thesis, and they build on one another: the failure diagnosis of RQ1 tells the data construction of RQ2 what to encode, the audited data feeds the one-API adaptation probe of RQ3, the controls of RQ4 establish what that adaptation actually changed, and RQ5 tests whether the controlled recipes survive five and then twenty-five jointly trained APIs. Every question is answered on the LibEvoBench completion benchmark [25] under the measurement protocol of Chapter 4, so the chain of evidence stays on one instrument from diagnosis to scale-up.

1. **RQ1: How does the base model fail on version-conditioned API completion, and what does it produce instead on the post-cutoff target APIs?**

   Before any adapter is trained, the baseline outputs on the broad benchmark sample are classified by the failure taxonomy of Chapter 4 into phantom, wrong-but-valid, anachronistic, and no-API outcomes, together with the substitutes the model prefers when it is wrong. The analysis then narrows to the post-cutoff target APIs, where the same classification shows what stands in for a symbol the model never produces, and a pre-declared family-proximity rate makes the nearby-substitute question mechanical for the controlled target. This measured structure decides what the training data of RQ2 must teach, while signature-level errors and variation across task types remain problem motivation outside the completed evidence.

2. **RQ2: Can source-grounded synthetic data encode the behavior needed to repair these failures, and how far can that adequacy be verified?**

   A teacher model works from each target API's signature, documentation, tests, and availability metadata, and RQ2 asks whether the corpora it produces demonstrably contain the target behavior, usage contexts, and decision boundaries that the RQ1 diagnosis calls for. The answer is read from the corpora themselves: coverage of the data roles and hard-negative families, the repetition that rendering and epoch expansion introduce, traceability to sources, validator outcomes with their applicable denominators, and prompt-answer overlap with the benchmark, all under the validation ladder of Chapter 4. Adequacy is claimed only as far as those checks reach, which keeps what the data contains separate from what a model later does with it.

3. **RQ3: Can a small LoRA adapter teach one post-cutoff API and reduce the phantom and substitute hallucinations that replace it?**

   The probe trains an adapter for one such API, `torch.compiler.is_compiling`, and asks whether the model then completes the benchmark's target rows with that API, stops producing the phantom and substitute answers that previously filled them, and leaves the rows where the API does not belong untouched. Acquisition is exact target match on the benchmark target rows and the diagnostic allowlist, leakage is reported as the labeled mention and resolved pair, and the paired movement metrics of Chapter 4 track what else changes on the same scored rows. RQ3 is answered

positively when target use rises while leakage, forgetting, and damage to already-correct rows stay measured and low.

4. **RQ4: Which training and control choices separate conditional target use from benchmark alignment, leakage, and interference?**

   A benchmark score can improve without any conditional target use, because fine-tuning may teach the output format, raise a general prior toward valid APIs, leak the new symbol onto rows where it is wrong, or damage behavior that was already correct. RQ4 therefore compares task formats, data compositions, matched format controls, and weight-space constructions in which everything but one ingredient is held fixed, so that each measured movement has an attributable source. Preference optimization and reinforcement learning remain discussed methods outside the completed evidence, for the reasons Chapter 7 develops.

5. **RQ5: Do the resulting recipes scale from one post-cutoff API to five and then twenty-five?**

   RQ5 moves from isolated target use toward the multi-API behavior a library-version specialist requires, and it adds what the one-API probe cannot show, competition among jointly trained targets and the cost of broader coverage. The five-API stage measures that competition together with variation across training seeds in a compact setting, and the twenty-five-API stage applies the frozen recipes, unchanged and under criteria fixed before launch, to twenty APIs they were never tuned on. The answer counts how many APIs improve, how gains vary across targets and seeds, and whether broader coverage raises leakage, interference, or retention cost, on denominators restricted to the rows the benchmark actually asks, which Chapter 5 fixes together with the cohort split.

Chapter 4 turns these questions into a measurement pipeline, and Chapter 6 answers them in order.

4

Methodology

4.1. Method Overview

Chapter 3 fixed the research questions, and this chapter defines the pipeline that answers them, summarized in Figure 4.1. We first approximate the versioned public API surface that a library documents for its users, the inventory from which candidate targets are drawn and against which predictions are checked for version validity. The base model's failures on the benchmark are then classified by an API-level taxonomy, and target APIs are selected only where the baseline shows a measured zero on the tested requests. A teacher model generates synthetic training data from source-grounded API metadata, documentation, signatures, tests, and controlled code contexts, with every row assigned an explicit role. LoRA adapters are trained on this data in two method families, called lanes throughout: direct SFT, which trains one adapter on the full data mixture, and task-arithmetic composition, which trains a matched pair of adapters, one with target-API pressure and one without, and composes their weight-space difference.

Every adapter from either lane is evaluated on the same matched benchmark rows for acquisition, leakage, and paired movement, because the experiments must distinguish production of a new API from conditional use in contexts where that API is correct, and the selected operating points are then checked for retention on control benchmarks outside the API benchmark. The protocol runs at one target API and scales to five and then twenty-five, with the concrete model, corpora, and target sets fixed in Chapter 5.

The dashed redesign loop in Figure 4.1 was driven by two signals: the paired-movement diagnostics

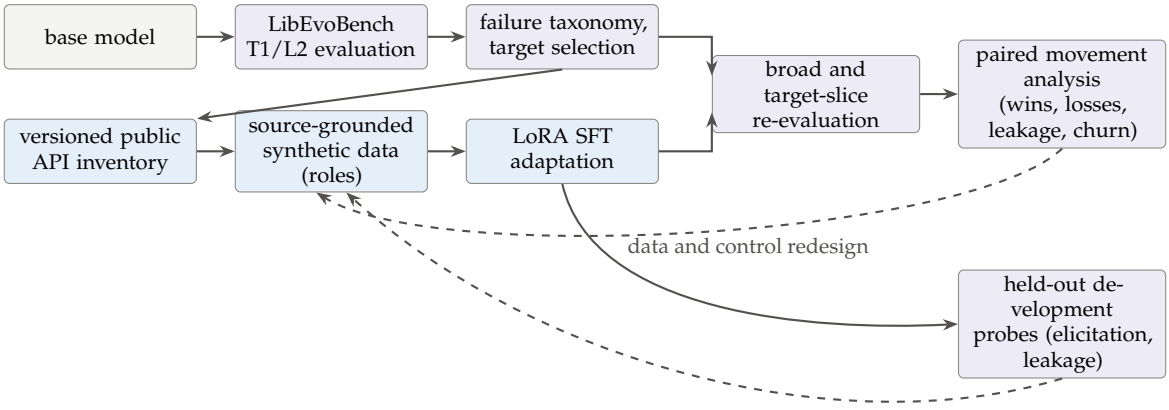


Figure 4.1: Methodological pipeline used to separate target API acquisition from benchmark-format learning, leakage, and non-target movement. Diagnosis on the benchmark selects the target and the failure categories, data construction encodes them as roles, and every adapter returns to the same matched-row evaluation and paired movement analysis. Two signals feed the dashed redesign loop, the paired-movement diagnostics on the benchmark side and the held-out development probes on the elicitation side.

on the benchmark side, and a small set of held-out development probes, prompts that lie outside the benchmark format and were never trained on, whose outcomes steered data revisions between generations.

The benchmark fills two roles that stay separate from the training data, serving as the measuring instrument for every reported claim and, through its target slice, as selection data under the protocol of Chapter 5. No benchmark row enters any training set, and the same row-level scoring protocol determines whether an adapter produces the target API, improves output compliance, leaks it onto rows where it is not the answer, or moves unrelated rows.

4.2. Benchmark Output Analysis and Failure Taxonomy

The benchmark used throughout is LibEvoBench [25], whose three task types (T1 masked-expression completion, T2 API identification from documentation, T3 signature production) and four context levels L0 through L3 are described in Section 2.2.1.

We use T1 because the studied failure is the choice of API identity inside code context, and we evaluate at L2 because it carries both the redacted documentation and the pinned-version requirement while still withholding the API name, the fullest rendering of the intended setting. L3 is excluded from acquisition claims because success there can be copying, and T2 and T3 are unused because documentation lookup and signature production are different constructs from the API-identity choice under study. Signature and argument checks are treated as a separate layer because an output can name the correct API while still using stale or invalid parameters.

The first analysis step turns generated benchmark responses into API predictions and classifies them against a versioned compatibility matrix. For the T1 completion rows, the scorer reassembles the completion with the code stem, the prompt's code prefix ending at the mask, removes any repeated prefix, and compares the dotted API path before the first parenthesis with the gold fill by string equality, an exact API identity match (`api_em`). Each predicted path is also resolved when possible and checked for existence in the target library version against the versioned API inventory, this check and the gold comparison together assign every scored row one taxonomy class, and the concrete prompt rendering and row samples are specified in Chapter 5.

Table 4.1 formalizes the failure modes of Chapter 1 as the outcome classes used throughout the thesis, adding the `no_api` class for rows that emit no resolvable API. A `wrong_api_valid` prediction names an API that exists in the target version but is not the one the row requires, so it is a wrong-but-valid benchmark error and stays outside the strict-hallucination categories, which contain only the `phantom` and `anachronistic` predictions that fail API validity directly.

The reported taxonomy has no separate unresolvable category. If the scorer cannot extract a versioned API path, the row is reported as `no_api`, while signature validity remains a separate validation layer.

The *family-proximity rate* counts predicted paths in the target's compiler namespace or the adjacent Dynamo and JIT introspection namespaces. We defined the rate before the analysis so that the nearby-substitute question of RQ1 stays mechanical. Availability metadata then decides whether a substitute is anachronistic, and qualitative inspection explains the cases the rate cannot distinguish.

Aggregate exact match is not sufficient for this thesis because target rows can be sparse inside a broad benchmark sample. A broad exact-match gain may come from non-target task calibration, fewer

Table 4.1: Failure taxonomy for API-level benchmark outputs.

Category	Definition	Interpretation
`correct`	Predicted API equals target API after normalization.	Exact API success.
`wrong_api_valid`	Predicted API exists in the target version but is not the target.	Wrong row-level choice, but not a strict hallucination.
`phantom`	Predicted API is absent from known version inventories.	Direct API hallucination.
`anachronistic`	Predicted API exists in another version but not in the target version.	Version-specific hallucination.
`no_api`	No resolvable API is emitted.	Output-contract failure, abstention, or task failure.

Table 4.2: Paired movement metrics used to interpret benchmark changes.

Metric	Definition
wins	Rows where the baseline is wrong and the adapter is correct.
regressions	Rows where the baseline is correct and the adapter is wrong.
net movement	Wins minus regressions.
churn	Rows where the prediction changes, regardless of correctness.
mention leakage	Non-target rows whose raw completion contains a trained target string.
resolved leakage	Non-target rows whose extracted prediction resolves to a trained target API.

`no_api` outputs, fewer phantom predictions, or row churn that is unrelated to the target API. For that reason, every candidate adapter is compared against the baseline on matched rows using the movement metrics in Table 4.2.

These metrics define the interpretation protocol for later chapters: a useful adapter should increase target recall while keeping mention leakage, resolved leakage, and regressions low, while a broad-score improvement with flat target recall is treated as consistent with task calibration or general API-selection movement. Taxonomy shifts are read beside exact match and the leakage pair, and an exact-match gain paired with falling phantom and anachronistic mass means that hallucinated output was replaced by correct API use. A rising anachronistic share is the taxonomy signature of leakage, the trained target landing on rows pinned to versions that predate its introduction.

4.3. Versioned Public API Inventory

4.3.1. Objective and Design Criteria

The goal of the extraction stage is to approximate the *public API surface* that developers are expected to use for a specific library version. For the downstream data-generation stage, we require the extractor to be *version-specific*, because the same library can expose materially different interfaces across releases, *precise*, so that synthetic examples are not dominated by semi-internal or developer-only objects, and *operationally robust* across heterogeneous third-party packages.

To select an extraction strategy, we compared four families of methods: runtime introspection, static source analysis with Griffe [45], static source analysis with AutoAPI [46], and Sphinx [47] inventory mining through the published `objects.inv` file. We used the comparison to decide which method yields the most suitable target set for version-specific synthetic data generation, with none of the four extractors taken as ground truth.

The aggregate extraction tables in this section are thesis-local analysis summaries from the public-API extraction comparison script family in the project repository, including the generic and pinned-version comparison drivers and their runtime, Griffe, AutoAPI, and Sphinx-inventory extractor wrappers, and in this chapter they are used as method-design evidence only.

4.3.2. Runtime Introspection as a Reference Extractor

As a broad reference, we implemented a runtime introspection procedure that operates on isolated environments pinned to exact library versions. For each target version, the procedure creates a clean environment, installs the requested package version together with minimal compatibility constraints when necessary, imports the target namespace, discovers public submodules, and recursively inspects reachable public objects together with their call signatures. When a library exposes multiple public namespaces, each namespace is inspected separately and the resulting API sets are merged.

For version-to-version comparison, extracted API paths are aligned by normalized name and their signatures are compared to identify added APIs, deleted APIs, and signature updates. Signature updates are further separated into changes affecting required arguments and changes affecting optional arguments. This procedure provides a useful approximation of the *runtime-discoverable* surface of a library version, but it is intentionally permissive: objects that are technically reachable at runtime can still be included even when they are not part of the maintainer-curated public interface. For that reason, runtime introspection is treated here as an upper-bound reference.

Table 4.3: Documentation-tooling prevalence among the filtered top-500 end-user-facing Python libraries. The conservative automatic-documentation category groups Sphinx, mkdocstrings, and pdoc.

Metric	Count	Share
End-user-facing libraries	468	100.00%
Sphinx users	295	63.03%
Conservative automatic-doc users	320	68.38%

Table 4.4: Normalized API counts produced by four extraction methods on the pinned-version comparison.

Library	Version	Runtime	Griffe	AutoAPI	Sphinx
`torch`	`2.2.0`	189,285	42,421	15,519	4,911
`numpy`	`1.26.4`	38,088	9,150	11,510	2,489
`matplotlib`	`3.8.4`	37,705	10,388	7,586	6,210

Table 4.5: Adequacy of the Sphinx-derived API set. Consensus denotes APIs that appear in at least three of the four compared extraction methods.

Library	Version	Precision vs runtime	Recall vs runtime	F1 vs consensus
`torch`	`2.2.0`	0.7776	0.0202	0.4766
`numpy`	`1.26.4`	0.8297	0.0542	0.4668
`matplotlib`	`3.8.4`	0.8775	0.1445	0.6561
Macro	N/A	0.8282	0.0730	0.5332

4.3.3. Documentation Tooling Among Popular Python Libraries

Before comparing extraction quality on pinned versions, we examined how common documentation-based API publication is among widely used Python libraries. After filtering the top-500 downloaded PyPI packages to 468 end-user-facing libraries, 295 libraries (63.03%) were found to use Sphinx, while 320 libraries (68.38%) used a conservative automatic documentation stack consisting of Sphinx, mkdocstrings, or pdoc. Table 4.3 summarizes these aggregate counts.

These numbers do not prove that every such library publishes a perfectly curated public API inventory, and the end-user-facing filter is itself heuristic, but they show that documentation-driven API publication is common enough to be a practically meaningful signal.

4.3.4. Pinned-Version Comparison and Final Method Choice

We then carried out the exact-method comparison on three representative pinned versions drawn from three widely used libraries: `torch==2.2.0`, `numpy==1.26.4`, and `matplotlib==3.8.4`. Table 4.4 reports the normalized API-set sizes produced by each extractor after filtering private and malformed paths.

Runtime extraction produces the largest surface in every successful case, reaching 189,285 APIs for `torch==2.2.0`, Griffe and AutoAPI are smaller but still typically much larger than Sphinx, and Sphinx inventory mining is always the smallest successful extractor. The counts differ because runtime and source-structure extractors enumerate broad reachable or source-visible surfaces, whereas Sphinx tends to approximate the explicit public interface that maintainers document for end users.

To quantify this distinction more directly, we compared the Sphinx-derived set against both the runtime extractor and a multi-method consensus set. The runtime comparison treats runtime introspection as a broad reference and therefore emphasizes whether Sphinx behaves like a conservative subset, while the consensus comparison defines a core API proxy as the set of paths appearing in at least three of the four compared methods, with the resulting metrics in Table 4.5.

Table 4.5 shows Sphinx to be consistently high precision with respect to runtime extraction (macro precision 0.8282) at low runtime recall (macro 0.0730): it behaves like a conservative subset of the much larger runtime-visible surface and cannot serve as an exhaustive extractor. Runtime discovery also depends on the host platform, and our extraction ran on a Linux CUDA machine, so it could not have discovered platform-specific surfaces such as Apple-Silicon-only PyTorch APIs. Despite being conservative, the Sphinx-derived set still captures a substantial portion of the API core shared across methods (macro F1 0.5332), a materially more favorable reading than the runtime comparison.

Each of the three pinned libraries exposes a different limitation of the alternative extractors. For `torch`, the comparison exposes severe surface expansion under runtime and source-based extraction, which is undesirable for downstream data generation because it would inject large amounts of semi-public structure. For `numpy`, the weak overlap between Sphinx and AutoAPI suggests that source-visible objects are not automatically a reliable proxy for the intended documented interface. For `matplotlib`, the Griffe run yields 10,388 normalized APIs, comprising 3,266 function entries and 7,122 method entries. Sphinx nonetheless remains the most conservative successful extractor for this library, which is desirable for downstream public-API targeting.

Appendix A provides a visual summary of both comparisons. Based on these observations, we chose Sphinx inventory mining as the public-API grounding signal for future library-version specialists. In the scoped experiments of this thesis, the extraction analysis plays a narrower role, motivating the choice of target APIs from maintainer-documented public surfaces instead of the much larger runtime-visible API graph. Runtime introspection is retained as a diagnostic reference during method comparison.

4.3.5. Downstream Use and Target Selection

The versioned inventory is used downstream to select public targets, to distinguish valid outputs from `phantom` and `anachronistic` outputs, to supply lifecycle labels for added or changed APIs, to enable hard-negative selection from semantically nearby valid APIs, and to provide signatures or parameter sets for static checks, making the inventory both a data-construction resource and an evaluation oracle.

We selected the controlled target `torch.compiler.is_compiling` using the following criteria:

- It is a documented public PyTorch API.
- It is post-cutoff or weakly represented in the base model.
- The baseline fails exact-match target diagnostics on it.
- Semantically nearby wrong alternatives exist, including compiler, tracing, and runtime-state predicates.
- It is narrow enough for manual inspection and mechanical validation.

4.4. Parameter-Efficient Adaptation with LoRA SFT

With the target selectable from the inventory above, the reported adaptation method is supervised fine-tuning with LoRA, in which the base model remains frozen and each adapted weight matrix receives a low-rank update:

$$W = W_0 + \frac{\alpha}{r} BA,$$

where A and B are trainable factors of rank r and α is a fixed scaling constant. The SFT objective minimizes negative log-likelihood on assistant or completion targets:

$$\mathcal{L}_{\text{SFT}} = -\sum_{t \in Y} \log p_{\theta_0 + \Delta_\phi}(y_t \mid x, y_{<t}),$$

where Y is the set of target-span token positions, x the rendered prompt, θ_0 the frozen base weights, and Δ_ϕ the adapter update.

Only tokens in the target response span are treated as the desired output, which makes the rendered task family, the prompt-and-answer format in which a training row is presented (Section 4.6.4), a methodological variable: each family trains the same API identity while placing different pressure on task alignment, output compliance, and transfer.

Candidate checkpoints are selected first on the target allowlist (every benchmark row whose gold answer is a target API, defined concretely in Chapter 5) and then by jointly inspecting target recall, leakage, broad non-target movement, and taxonomy shifts, because a checkpoint that maximizes target recall may also leak the target API onto unrelated rows, and this joint inspection makes paired movement analysis part of the method itself.

We treat preference and reinforcement-learning objectives as extensions in this chapter, and they contribute nothing to the reported empirical core, because DPO, KTO, GRPO, and DAPO are relevant to

this problem only when the reward or preference signal is API-level rather than string-level, a constraint that Chapter 7 develops into concrete reward requirements. Engineering efficient reward functions and setting up the infrastructure necessary for these fine-tuning methods is in itself an extensive research process, which is why we excluded them from the scope of this project.

4.5. Task Arithmetic as a Diagnostic Control

Task arithmetic [48] treats fine-tuning as movement in weight space. Training a model on a task moves its parameters from the base point to a new point, and the difference between the two, the task vector of that run, records everything the training changed. Task vectors compose to a useful approximation: adding one to the base model reinstates much of the trained behavior, adding several combines their behaviors, and subtracting one weakens or removes its behavior. Subtraction in particular gives a way to ask what one training run learned relative to another, because when two runs share everything except one ingredient, the difference of their task vectors cancels what both learned in common, to the extent the sharing is real, and keeps what the extra ingredient added.

We use this subtraction to separate two contributions that a single fine-tuned adapter mixes, benchmark-format alignment, which can raise broad scores without any target knowledge, and learning of the target API itself. We introduced it as a diagnostic methodology because we wanted to see whether we could separate knowledge injection from task alignment in the weight space and so measure the true knowledge injection more closely. The two adapters trained for this comparison are called arms, a target arm trained with target-API pressure and a control arm matched to task format and context where possible while excluding that pressure, so that what the arms learn in common should be the shared format and calibration content and what they do not share should be specific to the target pressure. Let θ_0 be the frozen base model, θ_t the target arm, and θ_c the control arm. Define:

$$\Delta_t = \theta_t - \theta_0, \quad \Delta_c = \theta_c - \theta_0, \quad K = \Delta_t - \Delta_c, \quad \theta(\lambda) = \theta_0 + \lambda K.$$

The difference vector K keeps the part of the target arm's movement that its control does not reproduce, and the composed model $\theta(\lambda)$ adds that part back to the base model at a chosen strength. The quality of this matching depends entirely on the chosen control data, and sweeping λ in the composed model $\theta(\lambda)$ measures how target acquisition changes relative to leakage, non-target movement, and broad exact-match shifts.

The implementation treats each arm's LoRA adapter as its effective weight delta and represents K exactly as a single stacked LoRA adapter of rank $2r$, concatenating the two factor pairs with the control factors carrying a negated scale, which introduces no SVD approximation. We call $K = \Delta_t - \Delta_c$, used unmodified and scaled only by λ at merge time, the plain difference vector.

Beyond the plain difference, a family of operator variants asks whether a more selective estimate of the target-specific direction improves the separation, each variant encoding one hypothesis about which coordinates or directions of the update are target-specific (Table 4.6). The first six transform the finished difference vector, while the two online forms, *gradient projection* and the *anti-leakage unlikelihood* term, apply the same idea during training itself.

The composition method reported in Chapter 6 is the plain difference vector. The operator variants ran under bounded settings on one frozen arm pair, where none of them improved on the plain difference's joint profile of acquisition and collateral movement, and tuning any of them into a competitive method would have required its own budget for truncation ranks, trimming and dropping rates, and Fisher estimation. We left that tuning outside the project's scope because the plain difference already performs the separation the diagnostic needs.

These variants change how the difference direction is estimated or filtered but not the interpretation boundary: all claims are relative to the chosen control, and reported task-vector conclusions never prove a universal API-knowledge subspace. Near-zero collateral movement is a diagnostic frontier and is not used as a selection objective, and couplings between acquisition and leakage are read with the caution that string-based leakage metrics can make that reading partly circular. Task arithmetic remains a diagnostic control throughout, with no claim that the composed models are deployable. Figure 4.2 sketches the geometric intuition behind the method: the target-arm delta Δ_t consists mostly of a large step along the direction it shares with the control arm, the format and calibration content both arms learn, plus a small residual specific to the target pressure. The difference vector K keeps that magnitude excess and residual while the shared base of the movement cancels, λ scales the kept part at merge time,

Table 4.6: Operator variants compared against the plain difference vector, each with its mechanism and the hypothesis it tests.

Variant	Mechanism	Hypothesis tested
per-module orthogonal projection	Removes from each module's target delta its component along the control delta, keeping only the orthogonal part.	Within each module, the target-specific content is orthogonal to the control direction.
rank truncation (SVD top-k)	Keeps only the top-k singular directions of the stacked update.	The meaningful difference concentrates in a few strong directions.
TIES-style magnitude trimming [49]	Zeroes low-magnitude coordinates and resolves sign disagreements between the deltas before merging.	Only large, sign-consistent coordinates hold target content, and small or conflicting ones accumulate error.
DARE-style random dropping [50]	Zeroes a random fraction of the difference's coordinates and rescales the survivors to preserve the expected update.	The useful content is redundant enough to survive sparsification.
Fisher-weighted subtraction [51]	Weights each coordinate by its estimated importance to the target loss under a diagonal Fisher approximation, with the contrastive variant weighting by the difference between target and control importance.	Importance to the target loss, absolute or relative to the control, marks the target-specific coordinates.
shared-subspace removal	Deletes the top-k output directions that the target update shares with the control update.	The shared output subspace holds common format content and the residual is target-specific.
gradient projection (online)	Projects each gradient step away from the control direction during target-arm training.	The separation can be built during training rather than recovered from the finished arms.
anti-leakage unlikelihood (online)	Adds an unlikelihood term that penalizes emitting the target in anchor contexts.	A training-time penalty on target emission in anchor contexts suppresses leakage at its source.

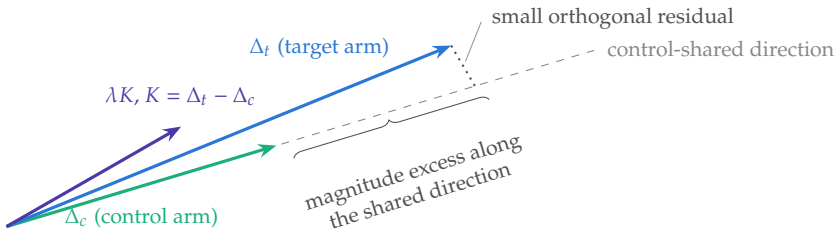


Figure 4.2: Geometric reading of the matched-control task vector within one arm pair. The target-arm delta consists mostly of a larger step along the direction shared with the control arm, plus a small residual. The difference vector K captures the excess, and λ scales it at merge time. The content of the shared direction (output format, sibling-API content, anchors) is a property of the chosen control, so all conclusions remain control-relative.

and because the shared direction's content is a property of the chosen control, the figure also shows why every conclusion drawn from K is control-relative.

4.6. Controlled Target API and Synthetic Data Design

The synthetic data design narrows the versioned public-API inventory to a single controlled target API, for which acquisition, leakage, and non-target movement can be measured row by row before any broader library-version adapter is attempted.

4.6.1. Target API Selection

Under the selection criteria of Section 4.3.5, `torch.compiler.is_compiling` becomes the empirical probe for the target-level narrowing of RQ1 and for RQ2–RQ4, a target on which four behaviors can be told apart: failure to know the target symbol, use of semantically nearby valid predicates, invention of

Table 4.7: Semantic training-row roles used to separate acquisition from leakage and task-format learning.

Data role	Purpose	Example signal	Main risk
target positives	Teach target API use.	Assistant emits `torch.compiler.is_compiling`.	Leakage.
anchors	Preserve nearby or stable APIs.	Assistant emits other Torch APIs.	Suppressing the target.
hard negatives	Define the decision boundary.	A sibling API is correct.	Weak or ambiguous negative.
format controls	Isolate task-format learning.	Same output shape, no target API.	Control-relative interpretation.

compiler-like phantom APIs, and leakage after training.

4.6.2. Grounding Sources and Training Inputs

All synthetic source examples, the target positives, anchors, and hard negatives, were generated by a Claude Sonnet 4.5 teacher model [31] given each target API's signature, documentation, and tests. Because LibEvoBench mines its completion rows from real public repository code, any real-world code example admitted into training could have coincided with a benchmark row and placed the measuring instrument inside the training data. We therefore generated every training example synthetically from those per-API sources, never from real-world code or benchmark rows, and the byte-exact and normalized overlap audits of Section 4.7 bound the residual risk. Deterministic builders then wrap the retained examples into the training task renderings, fill-in-the-middle (FIM), prefix completion, and code-gap chat, plus the benchmark-surface and matched target-control renderings at scale, so the model is trained only on the rendered prompts, code contexts, and assistant or completion spans on which the SFT loss is applied.

FIM trains completion of a missing middle span given the surrounding prefix and suffix, marked by model-specific sentinel tokens that the model must have learned during pretraining [52]. The sentinel vocabulary differs across model families, and newer models neither train nor document this interface, so FIM serves here only as a completion-format probe in the single-API experiments and is not a viable long-term data format for the instruction-tuned assistant setting this thesis studies.

4.6.3. Synthetic Data Roles

A row's role says what it teaches (target positive, anchor, hard negative, or format control, defined in Table 4.7), and its task family says how the row is rendered, so code-gap chat, for example, is a task family and not a role.

In the twenty-five-API control data, the control answer keeps the target row's import path and answer shape, so some control fills name an API that would not work in that context. These rows train only the format-matched control arm, and every composition claim stays relative to that control.

The hard negatives cover aliased imports, bound methods, constructors and factories, lifecycle and version confounds, neighboring namespaces, predicates and decorators, and receivers (the object a method is called on) and dimensions.

The roles are designed around two observable failure modes: positive-only endpoints may leak, emitting the target on off-target rows, while anchor-heavy mixtures may show suppression, failing to emit it on target rows. The conditional-activation regime in Figure 4.3 is defined by target acquisition together with low mention and resolved leakage. Chapter 6 places trained adapters in these regimes from their measured outputs, without attributing the behavior to a particular logit mechanism.

4.6.4. Task and Prompt Families

Task family varies independently of semantic role:

- FIM completion renders the row through the sentinel-marked fill-in-the-middle format of Section 4.6.2.
- Prefix completion continues the code from left context only.

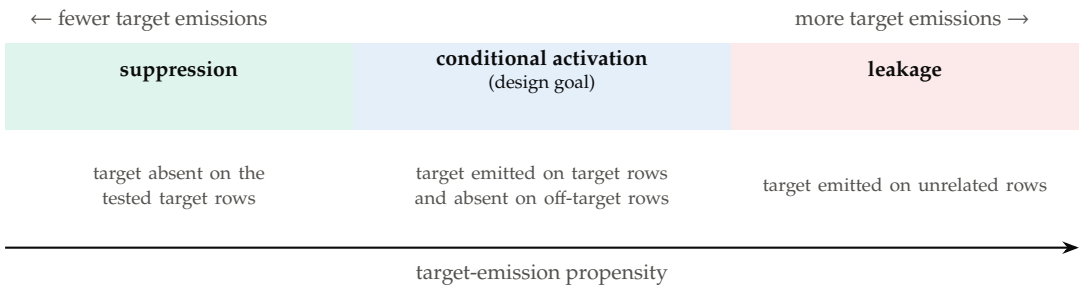


Figure 4.3: The three behavioral regimes used to classify trained endpoints. Suppression means that the target is absent on tested target rows, conditional activation combines acquisition with low leakage, and leakage means that the target appears on off-target rows. The horizontal axis is descriptive and does not identify a weight-space mechanism.

- Code-gap chat requests the missing expression or line through an instruction, with no documentation, signature, or target-path text in the prompt.
- Multi-task code chat renders the examples as replacement expression, API-call identification, guard-line patching, and semantic JSON outputs.
- Diverse-format chat includes benchmark bridges (benchmark-shaped completion rows mixed into otherwise diverse data), code comments, decision JSON, free code, grounded questions, and explanation or recall tasks.
- Benchmark-surface completion uses LibEvoBench-style fills, replacement expressions, and guard-line patches while excluding benchmark rows.

Staged anchor-then-target training reuses the same roles and changes only the training order, and the held-out development probes form a separate development split outside these task families.

4.6.5. Leakage Controls

The thesis uses leakage in three senses that these controls keep separate, and an unqualified leakage always means the behavioral sense. Behavioral leakage, the measured evaluation axis, is emission of a target API on rows where it is not the gold answer, reported as the labeled mention and resolved pair of Table 4.2 and formalized in Section 4.8. Data-side leakage is a training prompt that contains its own answer or identifying clues, the target path in the user text, documentation or signature fragments, or benchmark-template surface, and it would confound elicitation with knowledge because a model can copy a visible answer without having learned the API. Train-benchmark contamination is overlap between training rows and benchmark rows, the risk that the all-synthetic generation rule of Section 4.6.2 addresses at the source.

The construction-time audit addresses the data-side sense, checking every user prompt for direct target-path leakage, documentation leakage, benchmark-template leakage, and duplicate prompt forms, and the matching answer-side rules, answer-span placement of the target path and source-grounded hard negatives, are stated as validation rules in the next section. Contamination is bounded by the byte-exact and conservatively normalized train-evaluation overlap comparisons defined below, with no fuzzy or semantic near-duplicate detection applied, while behavioral leakage remains an evaluation outcome measured on the off-target rows.

4.7. Synthetic Data Validation

Whether a synthetic corpus is adequate for controlled training is hard to establish, because a training row is a prompt and answer span rather than a runnable program, no execution oracle covers the corpora at full scale, and a row can be correct in its syntax, in the API it names, or in its behavior under the pinned library version. A single pass-fail verdict would therefore merge claims with different evidence. The corpora must support adequacy for controlled training, teaching each target identity without leaking answers or copying the benchmark, which is a weaker claim than semantic perfection of every row. We therefore check each corpus on the three-tier ladder of Table 4.8, which runs from syntax through lexical API occurrence to runtime execution, and we pair the ladder with a direct audit

Table 4.8: Validation tiers for the synthetic corpora.

Tier	Validation	What it proves
1	Syntax.	The code-bearing answer parses, after prompt-context reconstruction where required.
2	Lexical API occurrence.	The designated full path or API leaf/suffix occurs on each applicable API-bearing row.
3	Runtime.	A row-level execution result records a clean pass for the row.

of the shipped corpora that re-runs the syntax and lexical tiers and binds its checks to the exact audited files. By design, lexical occurrence does not resolve imports, receivers, or runtime bindings, the runtime tier covers only part of each corpus, and the audit certifies the tiers it re-runs rather than executable semantics. Chapter 6 reports what each corpus passed under these definitions.

A tier that was not run is recorded as *not evaluated* rather than inferred from a later tier or from downstream model performance, and role balance, task-family coverage, duplicates, split groups, and train-evaluation overlap are separate corpus-level checks outside the ladder. Alongside these mechanical tiers and checks, we manually inspected the generated corpora at two granularities: per-example acceptance, the share of generated code examples accepted on manual reading for validity and quality, and per-dataset diversity adequacy, whether a generated dataset is judged sufficiently diverse in its scenarios and contexts. Both measures apply to the final generation methodology and teacher prompt, which we reached by revising earlier prompts until quality and diversity were acceptable, and Chapter 6 reports the acceptance and diversity rates recorded during our review. This reading is a subjective judgment that no mechanical tier certifies, and Chapter 7 discusses its subjectivity as a limitation.

Code-bearing answers are parsed with Python's abstract syntax tree parser, with fragment answers reconstructed inside their prompt context first so that a completion is not rejected merely for being incomplete in isolation. Lexical occurrence applies only to rows that designate an expected API, which excludes format-control rows because their correct answer substitutes another API, and the check accepts the full path or a matching leaf or suffix. Train-evaluation overlap is tested against all LibEvoBench T1 rows on the complete prompt-answer pair, compared byte-exactly and under a conservative normalization that lowercases text, removes code fences, and collapses whitespace, with the complete pair chosen as the criterion because an answer as short as an API path can legitimately recur across independently written examples. Exact row repeats are counted as well, so planned epoch repetition is never mistaken for independent source diversity.

Phantom or anachronistic negatives may be synthesized as training targets only when their invalidity is mechanically proven, since the teacher's own labeling is not accepted as proof, and hard negatives are valid only when the alternative API is source-grounded, version-valid, and correct for the negative context. Documentation and signatures may ground generation without appearing in the training prompt: when the goal is API identity learning, the API path must appear in the assistant or completion span so that the loss directly trains production of the target symbol.

4.8. Evaluation Methodology

We report on five surfaces (the broad T1/L2 sample, the broad target rows, the target allowlist, the broad non-target rows, and the retention harness), and Table 4.9 states the purpose and main metrics of each. The target allowlist contains every benchmark row whose gold answer is the target API, so the broad target rows are the subset of it that the sampled broad surface includes, and the allowlist is selection data for sweeps and checkpoints.

Signature checks form a validation layer outside the reported taxonomy and are not an evaluation surface in the completed evidence, so every completed evaluation claim in this thesis is an API-identity T1/L2 claim rather than an argument-validity claim.

Let a be the target API, T be target rows, N be non-target rows, B^+ be rows correct under the baseline, $U_{\text{raw}}(a)$ be rows whose raw generation contains the exact target string, and $U_{\text{res}}(a)$ be rows whose

Table 4.9: Evaluation surfaces and their methodological roles.

Surface	Purpose	Main metrics
broad T1/L2	API use in code context.	Exact match, taxonomy, target/non-target split.
broad target rows	Target API acquisition inside the benchmark sample.	Target recall.
target allowlist	Controlled acquisition diagnostic.	Exact target hits.
broad non-target rows	Locality and retention proxy.	Leakage, wins/losses, churn.
retention harness	Nine control benchmarks evaluated locally under official protocols (Section 4.8.1).	Paired per-item deltas, wins/losses, exact McNemar tests.

extracted prediction resolves to a. The reported row-level metrics are:

$$\text{target recall} = \frac{\#\{\text{target rows with exact target API } a\}}{|T|},$$

$$\text{mention leakage} = \frac{|N \cap U_{\text{raw}}(a)|}{|N|},$$

$$\text{resolved leakage} = \frac{|N \cap U_{\text{res}}(a)|}{|N|},$$

$$\text{leak-broken} = \frac{|B^{+} \cap N \cap (U_{\text{raw}}(a) \cup U_{\text{res}}(a))|}{|N|},$$

$$d_{\text{Coll}} = 100\left(\text{EM}_{\text{candidate},N} - \text{EM}_{\text{base},N}\right) \text{ pp}.$$

Mention and resolved leakage are reported as a labeled pair, and the two sets are not nested: the resolver can map an abbreviated or aliased extracted prediction to the target even when the raw completion omits the full target string, so resolved leakage may exceed mention leakage. The scale-up leak-broken metric uses the union in the equation above, while the legacy single-API mentions-breaking diagnostic counts baseline-correct rows in $U_{\text{raw}}(a)$ only and is labeled accordingly. The numerator of leak-broken, the count of baseline-correct off-target rows with a target mention or resolution, is also reported on its own as the broken baseline-correct rows of an operating point, the raw-count form in which the scaled selection rule of Chapter 5 prices damage against target hits. The d_{Coll} metric is a percentage-point delta on a fixed matched non-target sample rather than a raw count or an independently resampled benchmark estimate, and wins, regressions, and net movement are computed on matched rows as defined in Table 4.2. The aggregate taxonomy rates are the row fractions of each taxonomy class, with exact API match the `correct` fraction and the valid-API rate the `correct` plus `wrong_api_valid` fraction.

Target-string presence, a looser target-row diagnostic, checks whether the raw completion on a target row contains an API-specific leaf or alias substring recorded with the target set, for example `is_compiling` or `get_local_rank`. The check can reveal a target name that the exact scorer rejects, but it establishes no correct fill, parse, receiver binding, or use in context, so it is reported beside `api_em` and never treated as semantic correctness. Conversion, the exact-match conversion of an operating point, is the fraction of target rows passing the string-presence check whose completion the exact scorer also accepts, and a reported ratio whose denominator is empty, as conversion's is on a checkpoint with no string-presence rows, appears as N/A with its counts rather than being coerced to zero.

4.8.1. Retention and Control Benchmarks

Because an adapter that injects an API must not damage unrelated capabilities, retention outside LibEvoBench is one of the thesis's measured axes, and the retention harness of Table 4.9 consists of nine public benchmarks evaluated locally under their official protocols. We selected benchmarks to cover the capability families an adapter edit could plausibly damage, general knowledge, multi-step reasoning, grade-school mathematics, instruction following, general code generation, and data-science code, and we admitted only benchmarks with an official protocol rather than bespoke grading, with per-item outputs that support the paired base-model comparisons of Section 4.8.2, and with evaluation feasible

on local hardware. The nine benchmarks, with their protocols, published anchors, and citations, are listed in Chapter 5 (Table 5.2), and one lane, DS-1000, stores aggregate outputs only, so its comparisons stay descriptive.

4.8.2. Uncertainty and Paired Inference

Binomial proportions on the diagnostic target surfaces receive Wilson 95% score intervals [53]. We chose the Wilson interval because these surfaces are small, 12 broad target rows, 30 allowlist rows, 141 five-API rows, and 224 asked twenty-five-API rows, and because observed proportions lie at or near 0 and 1. There the standard normal-approximation interval degenerates, with zero width at $0/n$ or n/n and endpoints outside $[0, 1]$ near those extremes, while the Wilson interval keeps calibrated coverage [54]. The intervals describe the binomial uncertainty of one diagnostic proportion at a time and are not independent-samples tests.

Retention inference uses the two-sided exact conditional McNemar test [55] on discordant pairs. If w items are solved only by the adapter and l only by the base model, the null conditions on $w + l$ discordant items and assigns either direction probability one half. Holm adjustment [56] is then applied within each fixed arm-by-benchmark family specified in Chapter 5. Aggregate-only benchmarks remain descriptive because their stored outputs do not provide the paired correctness indicators required by this test.

4.9. Scaling Protocol

The protocol scales from one target API to multiple documented APIs by selecting APIs with a measured zero baseline, constructing per-API positives and hard negatives, adding stable anchors and matched controls, reporting per-API and macro metrics next to pooled ones, and measuring leakage and retention across non-target rows. The row roles retain their single-API meanings, while the scaled experiments add a prompt-exposure rule to the target gate, training rows matched to the benchmark's completion surface, one target-control composition over the full set, and replication.

Selecting zero-baseline target APIs. A target API is admitted only when the base model produces no exact-match completion on any issued target row at the context levels that withhold the API name: the code-only L0 surface and the L1 and L2 levels that add documentation and version context. The name-revealing L3 surface stays outside the gate for the same copying reason that excludes it from acquisition claims (Section 4.2). The gate establishes behavioral absence on the tested requests in the operational sense of Chapter 1, and the twenty-five-API extension exposed a blind spot in its original implementation, which recorded zero for rows that the L2 prompt builder had never issued, with Chapter 5 documenting the corrected asked-row denominator.

Matching training data to the benchmark surface. The scaled experiments make task format an explicit part of the data contract, building on the single-API experiments: rows intended to support scored completions reproduce the benchmark's completion pragmatics, including dangling receiver stems, aliased imports, distractor context, and continuation-only answers, while remaining synthetic. No benchmark row is copied, and a blocklist excludes benchmark-identifying receiver names and literals from generated rows. Chapter 6 measures how the resulting data generations change exact-match conversion without treating surface similarity alone as its cause.

Task arithmetic for the full target set. For multiple APIs, the matched-control construction of this chapter is applied at the level of the whole target set: one target arm and one format-control arm are trained jointly on all APIs, giving a difference vector of fixed rank independent of the number of APIs. The control coefficient defines $K_c = \Delta_t - c\,\Delta_c$, with $c \in [0, 1]$, and composition uses $\theta(\lambda, c) = \theta_0 + \lambda K_c$. The single-API diagnostics use full control subtraction ($c = 1$), while the scaled experiments sweep c and λ to measure the acquisition-leakage frontier without treating one coefficient's outcome as evidence for a binding mechanism. Composing per-API difference vectors trained in isolation is outside the protocol.

Pre-registration and replication. Because the scaled experiments iterate on diagnosed failures of earlier versions against the same benchmark, an evaluation-iteration threat, we declare each version's success and falsification criteria before launch, close falsified branches permanently, and re-train the

final configuration under two further seeds. Replication preserves the declared selection procedure: a peak-selected lane reselects a peak within each seed and reports the peak distribution, while a fixed-point lane compares the same checkpoint step or composition coefficients across seeds.

The empirical results of the five-API and twenty-five-API scale-ups are reported in Chapter 6 (Section 6.12).

5

Experimental Setup

Chapter 4 defined the diagnostic method, and this chapter applies it to LoRA adaptation of the instruction-tuned Qwen2.5 7B model on LibEvoBench at T1/L2, first for one API and then for target sets of five and twenty-five APIs. Checkpoints and compositions are selected on target rows (and, where a selection rule prices damage, on matched off-target rows), and the chosen operating points are then evaluated once on the retention suite of nine public control benchmarks (Section 5.2), which no selection rule read. These surfaces provide the acquisition, leakage, locality, and retention measurements reported in Chapter 6.

5.1. Base Model and Training Infrastructure

All reported experiments adapt `Qwen/Qwen2.5-7B-Instruct` [26]. We chose an instruction-tuned base because the benchmark uses a chat completion protocol (Section 5.2) and because the model is behaviorally absent on both single-API acquisition surfaces, with no exact target completion among the 12 broad target rows or the 30-row allowlist (0/12 and 0/30). Later exact matches on these requests measure acquisition relative to an observed zero baseline without establishing that the base weights contained no latent knowledge of the API.

The reported adapters are trained by supervised fine-tuning with LoRA [30], while the base weights remain frozen. The single-API format sweeps use Optuna, a define-by-run optimization framework [57], to enumerate bounded search domains and retain trial parameters with their evaluation results. Each trial calls LLaMA-Factory, a unified fine-tuning interface for language models [58], which keeps model loading, LoRA configuration, target-only loss, checkpoint export, and evaluation handoff common across the compared arms. We chose this combination to vary registered training factors without changing the trainer integration between trials. The five- and twenty-five-API experiments use the same training entry point through versioned shell runbooks that fix the data, training, evaluation, replication, and retention stages before launch. Preference optimization and reinforcement-learning objectives are outside the completed experiment lines and are discussed in Chapter 7, while LibEvoBench generation uses vLLM with greedy decoding at temperature 0 (Table 5.1).

Two same-weight runs of the full broad evaluation disagreed on 1.06% (122/11,485 rows) of the matched evaluation sample of Section 5.2 because of serving-side nondeterminism, which Chapter 6 treats as the broad-evaluation noise floor. Training used seed 42 unless stated otherwise, and the experiment logs record each run name, checkpoint, and artifact path alongside the corresponding result.

Table 5.1: Hardware and software environment retained with the reported adapters and repository lockfile.

Component	Recorded configuration
Hardware	Eight NVIDIA RTX 4090 GPUs with 24 GB each. Every training process used one GPU, while parallel runners assigned independent jobs to separate devices and staggered their starts.
Training stack	LLaMA-Factory 0.9.5.dev0 at commit `a132cc686b95`, PyTorch 2.8.0+cu128, Transformers 4.57.1, and PEFT 0.17.1.
Evaluation stack	vLLM 0.11.0 with greedy decoding, together with the repository's pinned LibEvoBench evaluator and benchmark database.
Orchestration	Optuna 4.5.0 runs the single-API sweeps, while versioned positive-anchor SFT and multi-API task-arithmetic runbooks define the scale-up experiments.

5.2. Benchmark and Evaluation Surfaces

All benchmark evaluation uses LibEvoBench [25] at T1/L2 under the task-type and context-level definitions of Section 2.2.1 and the scoring protocol of Section 4.2, restricted to `torch` releases 1.12.0 through 2.7.0. A system message defines the replacement task, one fixed few-shot turn establishes the output contract, each rendered prompt ends its code prefix at the mask and carries up to four lines of right context, and every generation is scored by the exact dotted-path match and failure taxonomy of Section 4.2.

Broad T1/L2 sample. The broad surface is a version-diversified subsample of the benchmark's full T1 table (170,532 torch completion rows) that keeps at most three code rows per API and version, giving 16,444 attempted rows, and the intersection of rows with generated completions across all compared models yields the 11,485 scored rows that serve as the main surface and as the matched sample for every paired comparison (Figure 5.1). Twelve of these rows have `torch.compiler.is_compiling` as the gold API, three per version for torch 2.4.0 through 2.7.0, and the remaining 11,473 rows are the non-target surface used for leakage, locality, and the benchmark-retention proxy.

Target allowlist diagnostic. The allowlist evaluation keeps every T1 row whose gold answer is the target API and lifts the per-cell sampling cap, which yields all 30 target rows the benchmark contains (4, 16, 6, and 4 rows for torch 2.4.0 through 2.7.0) and makes the 12 broad target rows a subset of these 30. Because it is cheap enough to run on every candidate checkpoint, it supplies the acquisition diagnostic and the selection data for sweeps and checkpoint racing under the protocol of Section 5.6. Of the 30 gold fills, 24 are the same guard-condition expression, so allowlist scores measure acquisition of that shape more than general usage diversity, and Chapter 6 interprets them with Wilson 95% intervals.

Held-out development probes. Nine synthetic chat prompts lie outside the benchmark's completion format: six target probes cover reverse documentation lookup, bug-fix diagnosis, and free-form code writing, with correct answers that must contain the target API, and three leakage probes require a sibling API such as `torch.jit.is_tracing`, making any target-API emission an error. The probes were written through the same generator pipeline as the training data rather than taken from real code, which could coincide with benchmark rows (Chapter 4), and they steered data revisions by testing whether a trained model could be elicited outside the benchmark format. Because they are few, substring-scored, and drawn from the same scenario universe as the training generator, they provide suggestive elicitation evidence only, no probe score is reported as a result, and no selection rule read them.

Retention / control benchmarks. The retention measurement of Section 4.8.1 uses nine public benchmarks that measure capabilities outside LibEvoBench, listed in Table 5.2 with each lane's local protocol, its role in paired inference, and the comparison of its local base score with the published Qwen2.5 report value [26]. These anchors serve only as protocol sanity gates that can expose a gross harness mismatch: MMLU, IFEval, HumanEval+, and MBPP+ agree with their anchors within item-sampling error, while GSM8K, MBPP, HumanEval, and MMLU-Pro carry harness-specific gaps between 1.2 and 5.2 percentage points in both directions (Table 5.2). Because these checks

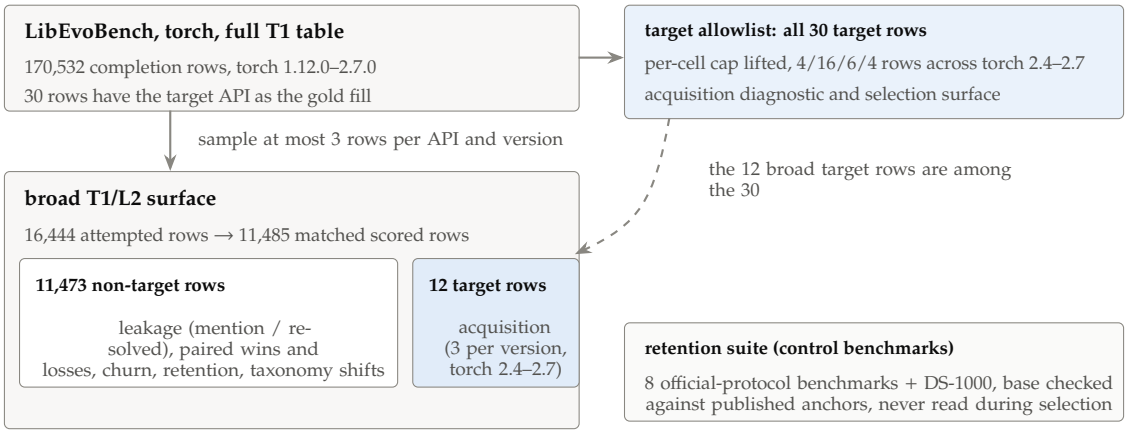


Figure 5.1: Evaluation surfaces of the single-API experiment line. The broad surface and the target allowlist are both drawn from the full T1 table, one by sampling at most three rows per API and version and one by keeping every row whose gold answer is the target API, so the 12 broad target rows are a subset of the 30 allowlist rows. Paired benchmark comparisons use the matched 11,485-row sample, and the retention suite uses published base-model scores as protocol sanity gates.

are not independent reproductions of the published environments, retention claims use paired deltas against the base model evaluated in the same local harness.

Base HumanEval and MBPP stay outside the paired families because their items nest inside the Plus variants and counting both forms would duplicate closely related tests, while DS-1000 contributes only a descriptive delta because its harness emits no per-item outcomes. The six remaining lanes enter every paired family, whose arm-by-benchmark composition is fixed in Section 5.6.

The completed retention corpus contains the selected operating points of four single-API recipes, four five-API arms, and three twenty-five-API arms, named in Table 5.8. Before each scale-up arm entered the suite, its merged model reproduced the recorded target score on the corresponding 141- or 291-row registered surface, linking the retention measurement to the artifact reported in Section 6.12.

The rerun measurements of Table A.1 fix the generation-noise floor against which Chapter 6 reads every retention delta. Eight same-weight, same-protocol serial rerun pairs across IFEval, GSM8K, and HumanEval+ changed no per-item outcomes, and two independent DS-1000 runs produced byte-identical generations on all 1,000 problems. This zero floor applies only to the measured model-protocol pairs, some of which use superseded task protocols, and does not establish determinism for every lane in the suite. On identical weights, a concurrent client changed about 6% of IFEval item outcomes, with aggregate shifts from −1.85 to +2.22 percentage points across the four single-API arms, so canonical IFEval deltas use serial runs while concurrent generative lanes retain this measured serving-noise limitation.

On the broad sample, the untuned base model scores 31.83% exact API match and 0% target acquisition on both diagnostics (0/12 broad target rows and 0/30 allowlist rows). These values define the Chapter 6 deltas, while Section 6.2 reports the full baseline taxonomy.

Table 5.2: The retention suite of nine public control benchmarks. Base gives the local score of the untuned base model under the stated protocol. Anchor gives the officially reported value: the Qwen2.5 report's `Qwen2.5-7B-Instruct` results for MMLU-Pro, GSM8K, IFEval, HumanEval, and MBPP, and its base-model MMLU value, which matches the local no-chat-template protocol [26], with the EvalPlus leaderboard supplying HumanEval+ and MBPP+ [59] and no official value existing for DS-1000 on this model. The four EvalPlus code lanes keep their task definitions unchanged under the chat pipeline.

Benchmark	Measures	Local protocol	Inference role	Base	Anchor
MMLU [60]	broad multiple-choice knowledge	5-shot log-likelihood	every paired family	74.2	74.2
MMLU-Pro [61]	harder multiple-choice reasoning, the suite's multi-step axis	5-shot chain-of-thought generation	every paired family	57.5	56.3
GSM8K [62]	grade-school math word problems	8-shot chain-of-thought, flexible answer extraction	every paired family	86.4	91.6
IFEval [63]	verifiable instruction following	zero-shot, prompt-strict	every paired family	71.5	71.2
HumanEval [64]	Python function synthesis	EvalPlus chat pipeline, greedy, base tests	secondary validation (items nest inside HumanEval+)	81.1	84.8
HumanEval+ [59]	HumanEval under extended test suites	EvalPlus chat pipeline, greedy, plus tests	every paired family	76.2	≈ 76
MBPP [59]	short Python programming tasks	EvalPlus chat pipeline, greedy, base tests	secondary validation (items nest inside MBPP+)	83.9	79.2
MBPP+ [59]	MBPP under extended test suites	EvalPlus chat pipeline, greedy, plus tests	every paired family	70.4	≈ 69
DS-1000 [65]	data-science code completion	BigCode completion pipeline, greedy	aggregate-only descriptive delta	28.2	n/a

5.3. Target API and Dataset Families

The single target API is `torch.compiler.is_compiling`, a public compiler-state predicate added in torch 2.3.0 and selected by the criteria of Chapter 4. Its synthetic training examples come exclusively from documentation, signatures, and controlled code contexts, with no benchmark rows and no real repository code used for generation (Chapter 4). The target path appears in the answer span, while the visible prompt contains only the information allowed by its training format.

The FIM and prefix datasets render the same 36 code examples as fill-in-the-middle completion, which supplies both prefix and suffix, or as prefix-only completion (Table 5.3). Anchor rows define contexts in which the target API must not be emitted by requiring nearby state predicates such as `torch.jit.is_tracing`, `torch.jit.is_scripting`, and `torch.is_autocast_enabled`, together with stable APIs such as `torch.tensor`, `torch.no_grad`, and `torch.manual_seed`.

The chat datasets contain a short instruction and a code snippet with a marked gap, optionally preceded by a `# requires` version comment, and the prompt contains no documentation or signature text. The multitask data renders 24 target and 24 anchor examples through replacement-expression, API-identification, guard-line, and semantic-JSON tasks.

The format-control dataset matches the multitask target arm in task families, row count, and training steps, but replaces the target API with eleven sibling predicates and applies no target pressure. The diverse-format datasets teach the API through natural-language description, code comments, free-form code, and decision-JSON rows without benchmark-style completion templates. Their held-out probes provided suggestive evidence that elicitation outside the benchmark format could change while exact benchmark emission remained flat, which motivated a bridge dose of about one quarter benchmark-shaped rows. The second version removes anchors and adds generic coding examples so that anchor suppression and format transfer are not changed together.

5.3.1. The Five-API Scale-Up Subset

For RQ5, we selected five public PyTorch APIs introduced after `torch` 2.0 for which the base model made no exact match on any issued target row at L0, L1, or L2. Table 5.4 lists the 141 target completion

Table 5.3: Training dataset families for the single-API experiments. Anchor rows require a nearby or stable Torch API and exclude the target API from the correct answer.

Family	Composition	What it tests
Positive-only FIM / prefix	12 target rows	Whether positives alone can inject the API (per task format).
Anchor FIM / prefix	36 rows: 12 target + 24 anchors	Whether anchor rows control leakage (per task format).
Chat, target-heavy (1:1)	24 rows: 12 target + 12 anchors (± version comment)	Chat learning from code-gap prompts.
Chat, balanced (1:2)	36 rows: 12 target + 24 anchors (± version comment)	Same, with stronger anchor pressure.
Staged anchor-then-target	anchor-only stage, then 12 target rows	Whether anchor pressure must stay active during target learning.
Multitask chat (task-vector target arm)	48 rows: 24 target + 24 anchors, four task families	Transfer of target knowledge across output contracts.
Format-control arm	48 rows over 11 sibling APIs, same task families, no target	Control estimating task-format and sibling-content movement.
Locality arm	target + hard negatives + 14 rehearsal APIs	Data-side locality reference.
Diverse-format (v1, v2) and bridge	natural-language, comment, free-code, decision-JSON rows (v2 ablates anchors and adds generic coding anchors)	Knowledge injection outside benchmark-shaped completion.

Table 5.4: The five-API zero-baseline scale-up subset. Target rows are T1 completion rows whose gold answer is the listed API, and the base model scores exact-match zero on all of them at context levels L0 through L2.

Target API	Shape	Target rows
`torch.compiler.is_compiling`	module-level predicate	30
`torch.compiler.disable`	module-level function	22
`torch.library.custom_op`	decorator	11
`torch.distributed.device_mesh.init_device_mesh`	module-level factory	33
`torch.distributed.device_mesh.DeviceMesh.get_local_rank`	method on a mesh receiver	45

rows drawn from the benchmark database. With the API name revealed at L3, a level excluded from the selection gate, the base model solved only 22.0% of the target rows (31/141), which identifies API selection as one component of the baseline failure without reducing every L2 failure to it.

The 141-row target surface defines the acquisition denominator, and within the 11,485 matched broad rows, 49 rows have one of the five APIs as their gold answer, leaving 11,436 off-target rows for leakage and locality measurements.

The strict benchmark matrix contains a median of 151 introduced public API names per post-2.0 PyTorch release, or 404 lifecycle changes when signature modifications are included. Conservative documentation filters across the studied libraries yield 57 to 89 introduced public-looking names per release. Five APIs represent roughly 3% of the strict PyTorch median and the twenty-five-API set of Section 5.3.3 represents roughly 17%, so both experiments remain multi-target probes below release scale.

5.3.2. Scale-Up Dataset Families

Positive-anchor chat SFT uses per-API packs of 24 to 32 target-positive scenarios, an equal number of hard negatives, and eight stable anchor rows, with the positives and hard negatives rendered under varied receivers and aliases, distractor APIs in the visible context, and version-pinned imports. Generated rows copy only the task's prompt shape, and a validator blocklist of benchmark surface tokens (extended at

Table 5.5: The twenty-five-API target set. Rows are registered benchmark rows, asked denotes rows queried at L2, and L3 gives baseline exact matches when the prompt reveals the API name (0 marks the strict zero-baseline subset). The last two columns give exact L2 matches of the two champions of Chapter 6: the SFT damage-priced champion (step 540) and the task-arithmetic champion (rank 32, seed 42). The five legacy APIs are marked ∘. The six APIs with zero asked rows are evaluated at L0, where the SFT champion answers 49 of their 67 rows (including all 36 `is_partial` rows) from a zero baseline.

API (under `torch.`)	**Family**	**Rows**	**Asked**	**L3**	**SFT**	**Task arith.**
`compiler.is_compiling`∘	compiler	30	30	12	24	29
`compiler.disable`∘	compiler	22	22	4	8	21
`compiler.set_stance`	compiler	3	3	3	3	3
`compiler.cudagraph_mark_step_begin`	compiler	4	4	0	4	1
`compiler.reset`	compiler	8	0	3	0	0
`distributed.device_mesh.init_device_mesh`∘	distributed	33	33	7	28	29
`...device_mesh.DeviceMesh.get_local_rank`∘	distributed	45	45	3	2	17
`...device_mesh.DeviceMesh.get_coordinate`	distributed	27	27	12	1	3
`...tensor.parallel.SequenceParallel`	distributed	2	2	0	2	2
`...tensor.placement_types.Placement.is_partial`	distributed	36	0	36	0	0
`library.custom_op`∘	library	11	11	5	10	10
`library.register_fake`	library	3	3	0	0	2
`export.load`	export	7	0	0	0	0
`export.ExportedProgram.named_buffers`	export	2	2	2	2	2
`export.decomp_utils.CustomDecompTable.update`	export	3	0	1	0	0
`backends.cuda.enable_cudnn_sdp`	backends	6	0	0	0	0
`backends.mha.set_fastpath_enabled`	backends	3	3	0	2	1
`nn.attention.flex_attention.BlockMask.from_kv_blocks`	nn	3	3	2	3	1
`nn.utils.clip_grads_with_norm_`	nn	2	2	0	0	2
`nn.utils.parametrizations.weight_norm`	nn	13	13	0	1	2
`xpu.current_stream`	xpu	2	2	0	2	2
`xpu.manual_seed_all`	xpu	4	4	1	4	2
`cpu.synchronize`	device utilities	12	12	12	12	12
`cuda.clock_rate`	device utilities	3	3	3	3	3
`get_device_module`	device utilities	7	0	3	0	0
Total		291	224		111	146

twenty-five APIs with the 291 target rows' 17 receiver identifiers and 3 dimension strings) rejects any generated row that reuses them.

The task-arithmetic lane trains a target adapter and a format-control adapter jointly across all five APIs. Each target row and its control counterpart use byte-identical user prompts, while the answer replaces the target with a pre-cutoff control API. The lane's fill rows are byte-checked against the benchmark template, receiver-binding rows cover the mesh method `DeviceMesh.get_local_rank`, and the control set is route-clean, with no control API reached through the import path of any target (for `custom_op`, whose import path is `torch.library`, a nearby control would share that route).

5.3.3. The Twenty-Five-API Target Set

The twenty-five-API experiments retain the validated five and add twenty post-2.0 APIs, giving 291 registered target rows. Of the 1,151 introduced names in the strict matrix, 146 have at least one benchmark row. The selection gates of Section 4.9 then require at least two rows, zero exact matches in the available L0 to L2 baseline cache, no attributable baseline emission of the name, no entry in the pre-2.1 documentation inventories, and a public name absent before `torch` 2.0, leaving twenty-five new candidates from which we selected twenty and registered five alternates. The five remaining alternates do not support another twenty-five-API increment under these filters, which would require more benchmark target rows and a new baseline cache.

The twenty new APIs span compiler, distributed, export, backends, xpu, neural-network, library, and device-utility families. Nine retain zero exact match at L3, even though that context reveals the API name, and Table 5.5 lists all twenty-five APIs with their per-API row, asked, and baseline counts.

Before launch, we registered criteria on four axes: transfer to the twenty new APIs, interference with the validated five (a 15-percentage-point bar on their 141 rows), locality, and adapter capacity. Each lane applied its frozen recipe to the twenty new APIs without per-API iteration, the final configurations were retrained under seeds 43 and 44 after their replication bands had been fixed, and the selected operating points entered the retention suite.

The L1/L2 prompt builder requires a non-empty documentation description and issues no request at those levels when a row fails the documentation-quality gate, which affected six targets and 67 of the 291 registered rows. Their selection-cache zeros at L2 reflected missing requests, while their L0 zero baseline was measured on issued requests. An acquisition rate over all 291 registered rows, or over all

150 registered new-API rows, would count unqueried rows as failures.

We retained the six APIs because only four of the five registered alternates pass the documentation gate, together providing eight L2-queryable rows, and swapping after the runs would have conditioned the target set on observed outcomes. The six remain training targets, while their evaluation claims are limited to measured L0 acquisition and leakage accounting on generated completions.

Chapter 6 calls the 224 issued L2 rows the *asked* surface, comprising 83 rows for the twenty new APIs and 141 rows for the validated five. L2 results use this denominator while keeping the 291 registered rows visible, and L2 acquisition for the six skipped APIs is undefined. Their acquisition is measured on the code-only L0 surface, with the complete results in Appendix A.6, and the selection script now rejects candidates without issued L2 requests. Within the matched broad sample, 103 rows have one of the twenty-five targets as their gold answer, leaving 11,382 off-target rows for leakage and locality measurements.

5.4. Training Configurations

Single-API task-format sweeps. The anchor FIM and anchor prefix experiments each ran a ten-trial grid over the shared search domain of Table 5.6, rendering the same 36 source examples with checkpoints at steps 72 and 144, so FIM and prefix form the budget-matched format comparison, while chat searched its own exploratory domain at lower learning rates and longer epoch budgets. Optuna used the 30-row target allowlist as its objective, after which representative checkpoints were evaluated on the broad surface for acquisition and leakage.

Fixed recipes. The single-API task-vector adapters (target, format control, and locality) share one recipe (Table 5.7), and each uses its final checkpoint, which keeps their weight deltas comparable and training length fixed across the comparison. In the five-API SFT lane, the matched schedule comparison holds the data and optimizer settings fixed while changing only the schedule, and the replication arms' 90-step cadence resolves the within-epoch peak. Every checkpoint of that lane receives a target evaluation with broad evaluation restricted to the registered candidate checkpoints, and each seed contributes its own maximum on the 141-row target surface, so this lane characterizes a distribution of per-seed peaks rather than fixed-step reproduction. At twenty-five APIs, the task-arithmetic recipe adds a rank-32 capacity arm that tests whether adapter capacity limits transfer and shares the rank-16 format-control adapter.

Data ordering. Both task-arithmetic scales train in one pass over an epoch-expanded parquet file, the planned passes materialized as repeated rows in one file, with shuffling disabled and the target data arranged in API-blocked optimizer windows, rows grouped so that consecutive optimizer steps see one API. The twenty-five-API SFT lane also disables shuffling and consumes a parquet file containing two pre-ordered logical passes in one trainer epoch. We deliberately chose this ordering to stabilize training, because our earlier runs showed that mixing rows from multiple APIs in the same batch (which regular random shuffling produces) increases training instability under the task-arithmetic method.

Operating-point selection. Seed 42 supplies the twenty-five-API SFT checkpoint-selection trace. Because none of the ten broad-evaluated checkpoints of the twenty-five-API arm met the registered

Table 5.6: Registered search domains of the single-API sweeps. Cells give the fixed value or the swept set, and n/a marks a value the recorded domain leaves unstated.

Dimension	Anchor FIM / prefix grid	Chat search
Learning rate	$\{5, 10, 15, 20, 30\} \cdot 10^{-5}$	$1{\cdot}10^{-5}$ to $8{\cdot}10^{-5}$
Epochs	2 or 4	4, 8, or 12
LoRA rank	16	16 or 32
Dropout	0	0 to 0.05
Schedule	cosine	constant or cosine
Per-device batch	n/a	2
Gradient accumulation	n/a	2 or 4

Table 5.7: Fixed training recipes. Every recipe adapts the seven projection modules (`q`, `k`, `v`, `o`, `gate`, `up`, `down`) with sequence cutoff 4096 and reaches effective batch 8, and batch cells give per-device size × gradient accumulation where recorded.

Arm(s)	LoRA	LR, schedule	Batch	Duration, checkpoints	Seeds
Single-API task-vector adapters (target, format control, locality)	r 16, α 32	$5{\cdot}10^{-5}$, constant	2×4	three epochs, final checkpoint	42
Five-API direct SFT	r 64, α 128, dropout 0	$5{\cdot}10^{-5}$ peak, constant and cosine (warmup 0.1)	effective 8	six 180-step epochs, saved every 180 steps (original arms) or 90 (replication)	42, 43, 44
Twenty-five-API direct SFT	r 64, α 128, dropout 0.05	$2.5{\cdot}10^{-5}$, constant, no warmup	2×4	one trainer epoch, saved every 180 steps (five-API interference control every 90)	42, 43, 44
Five-API task-arithmetic adapters (target, format control)	r 16, α 32, dropout 0.05	$2.5{\cdot}10^{-5}$, constant	2×4	one pass over the epoch-expanded parquet	42, 43, 44
Twenty-five-API task-arithmetic adapters	r 16, α 32 (target, control) and r 32, α 64 (capacity), dropout 0.05	$2.5{\cdot}10^{-5}$, constant	2×4	one pass over the epoch-expanded parquet	42, 43, 44

ceiling of 60 broken baseline-correct rows, the registered fallback maximized target hits minus half that broken-row count and selected step 540, whose count of 64 was the observed floor. Seeds 43 and 44 then repeat the frozen recipe and are compared at step 540, while their separate peaks define only the acquisition envelope. Five-API direct SFT reports the acquisition peak selected separately within each seed, twenty-five-API direct SFT uses this seed-42 damage-priced selection (maximizing target hits under a cap on broken baseline-correct rows) followed by fixed-step replication, and task arithmetic evaluates composition points whose coefficients were registered before the broad comparisons. The operating point a lane selects and reports in this way is called its champion.

5.5. Task Arithmetic Setup

The single-API target and format-control adapters start from the same frozen base checkpoint and use the fixed recipe above. Following task arithmetic [48], each adapter defines an effective weight delta $\Delta = sBA$, where $s = \alpha/r = 2$, from which the exact stacked difference adapter of Section 4.5 is built. Candidate models $\theta_0 + \lambda K$ use $\lambda \in \{0, 0.5, 1, 1.25, 1.5, 2\}$, and $\lambda = 0$ reproduces the base model for the canonical collateral reference. The compared operators are the plain difference and the operator variants of Section 4.5, each measured on the target allowlist, the broad target rows, and the broad non-target rows.

At five APIs, one target adapter and one format-control adapter are trained jointly across all APIs, and their rank-16 source adapters form an exact rank-32 vector independent of the number of targets. The registered grid covers control coefficients from 0 to 1 and scales from 0.5 to 1.3, with $c = 0.25$ retained at two final operating points: the acquisition point at $\lambda = 1.0$ and, at $\lambda = 0.9$, the knee, named because it sits just below the bend of the acquisition-leakage curve where leakage falls faster than acquisition.

The 25-API grid uses four registered (c, λ) pairs: $(0, 1.0)$, $(0.25, 0.9)$, $(0.25, 1.0)$, and $(0.5, 1.0)$. The rank-16 target and control adapters form exact rank-32 vectors, while combining the rank-32 target capacity arm with the shared rank-16 control forms exact rank-48 vectors. In both scale-up settings, subtraction is defined relative to the chosen format control, and c sets the measured acquisition-leakage tradeoff.

5.6. Metrics and Reporting Protocol

- Acquisition and the mention and resolved leakage rates follow the definitions of Chapter 4, and leakage always appears as a labeled pair.
- Correct, wrong-but-valid, phantom, anachronistic, and no-API rates are reported from the row-level failure taxonomy, with shifts computed on the same matched rows as the exact-match comparisons.
- Wins, regressions, net movement, churn, and d_{Coll} use the matched 11,485-row universe, the target-set-specific off-target subset, and the same baseline parquet. The measured d_{Coll} noise floor is ±0.06 percentage points, and Chapter 6 leaves d_{Coll} changes below that floor and same-weight row differences within the 1.06% serving churn uninterpreted.
- The 12 broad target rows and the 30-row allowlist that contains them are diagnostic samples. Allowlist proportions receive Wilson 95% intervals when compared, and a one-row difference does not support a method effect.
- Under one task-vector training configuration, allowlist exact match varied from 50.0% to 70.0% across three runs (15/30, 20/30, and 21/30). Cross-run gaps are read against this spread, while comparisons that reuse one trained adapter pair and vary only the operator or λ are structurally matched.
- Every retention arm is compared with the base model item by item under the multiple-comparison families fixed below, with Chapter 6 reporting newly solved and newly failed items beside each delta and interpreting each result against the lane-specific serving-noise measurements of Section 5.2.
- Five-API acquisition rates use the 141-row target surface, and leakage rates use the 11,436 off-target rows. Counts remain visible beside percentages, and the macro mean accompanies the micro rate because the five per-API denominators are unequal.
- Every 25-API acquisition result reports the rate on the 224 rows the benchmark asks at L2, with the 291 registered rows visible. The new cohort uses 83 asked rows, the validated-five cohort uses 141 rows, and the six skipped APIs have L0 results only, while leakage and locality use 11,382 off-target rows. For the seed-42 SFT25 selection trace, the winner rule first maximizes target hits among checkpoints with at most 60 broken baseline-correct rows and, if that set is empty, maximizes target hits minus half the broken-row count.
- Five-API SFT replication reports the mean and range of the three per-seed peaks, with the seed-42 record labeled a *single-seed maximum*. SFT25 fixes the seed-42-selected step 540 before evaluating seeds 43 and 44, and reports separate per-seed peaks only as an acquisition envelope. Task-arithmetic replications hold the registered control coefficient and scale fixed.

We fixed four multiple-comparison families before interpreting the results, over the arms named in Table 5.8. The single-API family contains Chat, target-heavy (1:1), version comment, Staged target t0, Multitask chat t13, and Difference vector $\lambda = 1.25$ across the six pairable benchmarks (24 tests). The five-API family contains the SFT record and low-leak arms together with the task-arithmetic champion and knee across the same six benchmarks (24 tests). The SFT25 champion forms one six-test family, while the TA25 champion and knee form a twelve-test family. Each cell is tested with the exact conditional McNemar and Holm procedure of Section 4.8.2 within its fixed family. Base HumanEval and MBPP, together with the aggregate-only DS-1000 lane, stay outside these families (Section 5.2).

Selection protocol and benchmark reuse. Sweep objectives, checkpoint racing, and operating-point choice read benchmark target data: the 30-row allowlist at one API, the 141-row surface at five APIs, and the 224 asked rows at twenty-five APIs. Broad matched rows also enter selection when a rule prices leakage or broken baseline-correct rows, including the twenty-five-API SFT damage rule. These target and broad magnitudes are therefore selection estimates rather than held-out performance. We accepted this reuse because the benchmark contains too few target rows for a split with useful power and because exact API emission in repository code is the quantity under study. The zero baseline establishes the

Table 5.8: Display names of the eleven operating points that enter the retention suite and the multiple-comparison families. Tags of the form t*N* identify individual checkpoints of the corresponding single-API sweep.

Display name	Lane	Operating point
Chat, target-heavy (1:1), version comment	single-API SFT	selected checkpoint trained on the target-heavy chat dataset (12 target and 12 anchor rows) with the `# requires` version comment (Table 5.3)
Staged target t0	single-API SFT	staged anchor-then-target arm at its checkpoint t0
Multitask chat t13	single-API SFT	multitask chat arm at its low-leakage checkpoint t13
Difference vector $\lambda = 1.25$	single-API task arithmetic	plain difference adapter applied at scale $\lambda = 1.25$
SFT record	five-API SFT	seed-42 cosine-schedule checkpoint at step 360, the single-seed acquisition maximum
SFT low-leak	five-API SFT	constant-schedule checkpoint at step 960 with the smallest leakage pair and fewest broken baseline-correct rows among the lane's evaluated operating points
Task-arithmetic champion	five-API task arithmetic	composition at $c = 0.25$, $\lambda = 1.0$
Task-arithmetic knee	five-API task arithmetic	composition at $c = 0.25$, $\lambda = 0.9$
SFT25 champion (step 540)	25-API SFT	seed-42 damage-priced selection at step 540
TA25 champion	25-API task arithmetic	rank-32 composition at $c = 0.25$, $\lambda = 1.0$
TA25 knee	25-API task arithmetic	rank-16 composition at $c = 0.25$, $\lambda = 0.9$

direction of acquisition on the tested requests, while the same-configuration spread of 15 to 21 hits out of 30 and the single-seed-maximum label quantify some of the resulting selection uncertainty.

For SFT25, the step-matched retrainings under seeds 43 and 44 fix the selected recipe and checkpoint step before evaluation, while the five-API SFT replication selects one peak per seed and estimates peak variability. The twenty-new-API phase applies a pre-specified recipe under the same instrument. Neither design creates an independent held-out benchmark, but each makes its selection dependence explicit. The retention suite is the only completed evaluation family that no acquisition selection rule read. FIM and prefix share source examples and a search grid, while chat uses its separate exploratory search space, so direct format comparisons are limited to the matched FIM-prefix pair.

6

Results

6.1. Scope and Headline Claims

This chapter reports the completed empirical evidence of the thesis: the controlled single-API experiment line for `torch.compiler.is_compiling` and the scale-up line of RQ5, first to five and then to twenty-five post-cutoff APIs. The results proceed in the order needed to interpret the intervention: baseline failure, direct evidence about the synthetic corpora, single-API adaptation and controls, five-API and twenty-five-API scale-up, then retention outside the benchmark.

The headline finding, developed over the following sections and collected in Section 6.14, is that a small LoRA SFT adapter *can* make a missing post-cutoff API producible, that the harder problem is conditional use without leakage or collateral movement, and that this distinction persists at five and twenty-five APIs, with the scale-up evidence remaining tied to the same evaluation instrument. Unless stated otherwise, every number in this chapter is computed on the matched 11,485-row LibEvoBench T1/L2 sample of Chapter 5 [25], on the measurement surfaces and denominators collected in Table 6.1. Leakage is always reported as the pair (mention, resolved), the two indicators are non-nested, and scale-up leak-broken rates use their union as defined in Chapter 4. Target-surface rates are diagnostics on the selection surface of Chapter 5, reported with the Wilson 95% intervals of Section 4.8.2 where a comparison is implied, and the held-out development probes contribute no headline number. Supplementary per-run and replication views of the same data are collected in Appendix A.

Table 6.1: The measurement surfaces of this chapter. All rows are LibEvoBench T1 completions at context level L2. The single-API surfaces partition the matched 11,485-row sample, the scale-up target surfaces replace the single-API ones at their scale, and rows the benchmark never asks at L2 (67 of the 291 registered twenty-five-API rows) enter no rate.

Scale	Surface	Rows	Reported as
Single API	broad target rows	12	exact acquisition (diagnostic)
Single API	target allowlist	30	exact acquisition (diagnostic)
Single API	non-target rows	11,473	leakage pair, collateral movement, churn
Five APIs	target surface	141	exact acquisition
Five APIs	off-target rows	11,436	leakage pair, leak-broken rows
Twenty-five APIs	asked target rows	224 of 291	exact acquisition (83 in the new cohort)
Twenty-five APIs	off-target rows	11,382	leakage pair, leak-broken rows

6.2. Baseline Failure (RQ1)

On the full matched broad sample the base model reaches 31.83% exact API match, and Figure 6.1 shows its per-version profile next to the outcome-taxonomy composition of Chapter 4. Wrong-but-valid choices and phantom APIs carry almost equal shares of the output, phantom predictions alone account for 39.97% of the erroneous rows (3,129/7,829), and the anachronistic share is close to zero before any adaptation. Exact match varies by about eight percentage points across the ten torch versions, and on the four versions that contain target rows the base model completes none of them with the target API.

The target API line itself starts from a complete behavioral absence, with 0% (0/12) exact matches on the broad target rows, 0% (0/30) on the target allowlist, and mention and resolved leakage both at 0% (0/11,473) on the non-target rows, which establishes the absence, in the operational sense of Chapter 1, from which acquisition is measured.

Table 6.2 lists the baseline predictions on the twelve broad target rows together with their outcome classes and the family-proximity rate of Chapter 4: every wrong answer is a semantically nearby check of compiler, tracing, CUDA, or runtime state. The scorer classifies 58.3% of these rows (7/12) as wrong-but-valid choices of existing runtime checks and 41.7% (5/12) as phantom names that exist in no inventory version, two of which (`torch.is_compiling`, `torch.dynamo.is_compiling`) place the target's own leaf under a wrong module path, and none as anachronistic. The majority lie inside the compiler, Dynamo, and JIT introspection namespaces while the rest name CUDA hardware-availability checks, so the predictions have the expression type required by the context, a runtime-state check, while missing the post-cutoff symbol that provides it.

The same structure holds across the full broad sample, where Table 6.3 lists the most frequent wrong-but-valid confusion pairs and phantom names. The leading confusions replace the gold API with a semantic sibling from its own family (`torch.chunk` completed as `torch.split`, `torch.enable_grad` as `torch.no_grad`, `torch.jit.trace_module` as `torch.jit.trace`) or with a deprecated ancestor (`torch.linalg.svd` as the deprecated `torch.svd`), and the leading phantom names attach real functionality to fabricated paths, usually truncating the true module path (`torch.DataLoader` for `torch.utils.data.DataLoader`, `torch.empty_cache` for `torch.cuda.empty_cache`, `torch.load_state_dict` for the `Module` method) or stopping at a bare namespace (`torch.cuda`). Anachronistic predictions are rare, 0.36% of the scored rows (41/11,485), and nearly half of them (19/41) name a deprecated linear-algebra solver (`torch.eig`, `torch.lstsq`, `torch.symeig`) on rows whose pinned version postdates its removal. Baseline failure on this benchmark is therefore structured substitution among semantic neighbors and path-plausible fabrication, and version misplacement plays almost no part in it.

The zero-baseline failure and its substitution structure repeat on the scale-up target frames. On the 141-row five-API surface of Chapter 5, the base model solves 0% (0/141) and produces 51.1% phantom, 34.8% wrong-but-valid, 14.2% no-API, and no anachronistic outputs. On the 224 twenty-five-API rows that the benchmark asks at context level L2 (Section 5.3.3), it solves 0% (0/224), with 44.2% phantom, 29.9% wrong-but-valid, 24.1% no-API, and 1.8% anachronistic outputs (4/224, two distinct

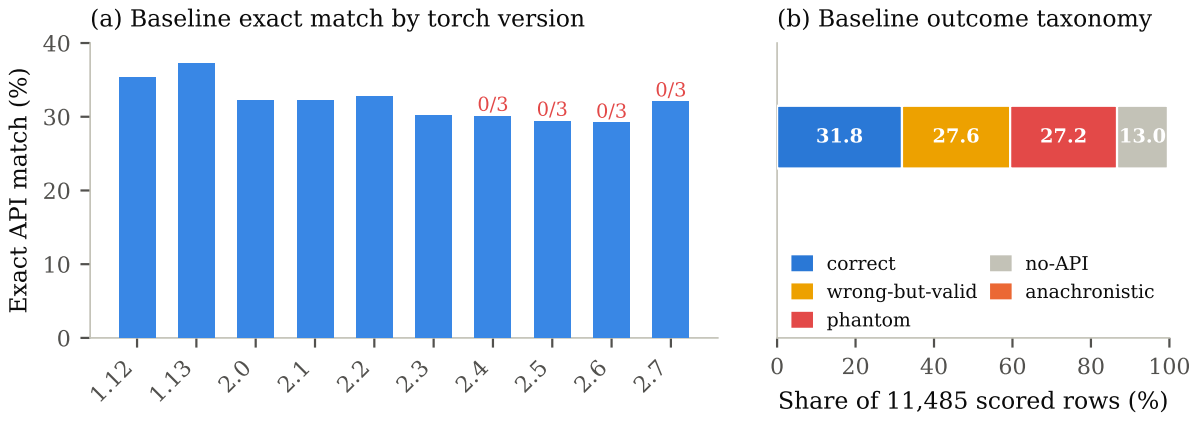


Figure 6.1: Baseline behavior on the matched 11,485-row broad sample. (a) Exact API match by torch version, where the red annotations mark the four versions that contain the 12 target rows, none of which the base model completes with the target API. (b) Outcome taxonomy over all scored rows: 31.83% correct, 27.58% wrong-but-valid, 27.24% phantom, 12.99% no-API, and 0.36% anachronistic.

Table 6.2: Base-model predictions on the 12 broad target rows for `torch.compiler.is_compiling`, with the outcome class of Chapter 4 and the family-proximity decomposition. All errors are semantically nearby checks of compiler, tracing, CUDA, or runtime state, none is the target API, and none is anachronistic.

Baseline prediction on target rows	Rate (of 12)	Class
`torch.cuda.is_available()`	33.3% (4/12)	wrong-but-valid
`torch.jit.is_scripting()`	25.0% (3/12)	wrong-but-valid
`torch._is_cuda`	8.3% (1/12)	phantom
`torch.dynamo.is_compiling():`	8.3% (1/12)	phantom
`torch.is_compiling()`	8.3% (1/12)	phantom
`torch.is_dynamo_available()`	8.3% (1/12)	phantom
`torch.is_dynamo_enabled()`	8.3% (1/12)	phantom
Substitute rates		
Family proximity (compiler, Dynamo, JIT)	58.3% (7/12)	
CUDA availability checks	41.7% (5/12)	

Table 6.3: The most frequent baseline errors on the matched 11,485-row broad sample: the eight leading gold-to-prediction pairs among the wrong-but-valid rows and the eight leading phantom predictions, with absolute row counts. All names are under `torch.`

Wrong-but-valid confusion (gold → predicted)	Rows	Phantom prediction	Rows
`linalg.svd` → `svd`	28	`load_state_dict`	54
`distributed.monitored_barrier` → `distributed.barrier`	23	`cuda`	41
`as_tensor` → `tensor`	20	`empty_cache`	35
`atan` → `atan2`	19	`random_split`	25
`chunk` → `split`	18	`DataLoader`	24
`enable_grad` → `no_grad`	18	`fft`	22
`nn.ModuleList.extend` → `nn.ModuleList.append`	16	`size`	22
`cholesky_inverse` → `inverse`	15	`synchronize`	21

predictions). The confusions keep the single-API structure: `torch.compiler.disable` is completed as `torch.no_grad` on 16 of its 22 rows, the mesh method `get_local_rank` as its sibling `get_rank` on 12 of 45, and `torch.compiler.is_compiling` as `torch.cuda.is_available` on 7 of 30, while the leading phantoms again truncate or misplace real paths (`torch.DeviceMesh` on 17 rows for the class under `torch.distributed.device_mesh`, `DeviceMesh.current_device` on 11, `torch.custom_op` on 6 for `torch.library.custom_op`).

Because errors on every tested frame concentrate in semantic neighborhoods and path-plausible fabrications while phantom mass is already high across the whole benchmark, deciding whether an intervention teaches a target or merely shifts errors within its neighborhood requires the target, leakage, and taxonomy measurements together. For the controlled target, these structured API-identity failures answer the target-level half of RQ1: the substitutes concentrate in the target's semantic neighborhood. The same substitution-and-fabrication profile carries to the five-API and twenty-five-API frames, while signature validity and task-by-context variation remain untested.

6.3. Direct Synthetic-Data Evidence (RQ2)

RQ2 asks whether the synthetic data are adequate for controlled adaptation. Table 6.4 reports the tiered audit of Section 4.7 over the twelve citable or qualified corpus families, whose 25,816 structural rows include reused renderings of the same source material, so every total below counts audited rows, not independent examples. The rendered-input registry separately binds every citable quantitative arm of this chapter to its exact training files, 58,716 rendered rows in 24 artifact groups and 35 files across the one-, five-, and twenty-five-API lanes. Some final target-control files appear in both views, so the structural and rendered row counts must not be added. Five corpus families additionally carry *live source contracts*, checks re-run on the shipped files that confirm each corpus still contains its intended roles and row counts.

Two of the table's findings shape how the corpora may be read. Complete prompt-answer pair overlap against all 170,532 rows of the full T1 benchmark is zero in both views under exact and normalized comparison, and although individual answer strings recur because API calls are short, no complete training instance duplicates a benchmark instance. The exact repeats, concentrated by design in the

Table 6.4: Direct RQ2 audit. The structural and exact rendered-input views overlap, so they remain separate. Runtime results are reported at their achieved coverage.

Audit check	Result	Coverage	Interpretation
Python AST parsing	100%	25,800/25,800	All code-bearing rows parse
Lexical API occurrence	100%	14,972/14,972	Full path or leaf/suffix, with format-control rows excluded because their expected output substitutes another API
Live source contracts	5/5 pass	240/240, 320/320, 320/320, 1,600/1,600, 1,600/1,600	Exact row agreement for all five live per-corpus validators
Within-corpus exact repeats	62.58%	16,155/25,816	Planned epoch expansion: 75.19%, 75.06%, and 75.05% of the three task-arithmetic pairs
Structural complete-pair overlap	0%	0/25,816 vs 170,532 benchmark rows	Zero exact and normalized matches
Rendered complete-pair overlap	0%	0/58,716 vs 170,532 benchmark rows	24 artifact groups and 35 training files, zero exact and normalized matches
Single-API positive runtime	100%	12/12	Complete for this source corpus
Positive-plus-anchor runtime	33.3%	12/36	Partial coverage only

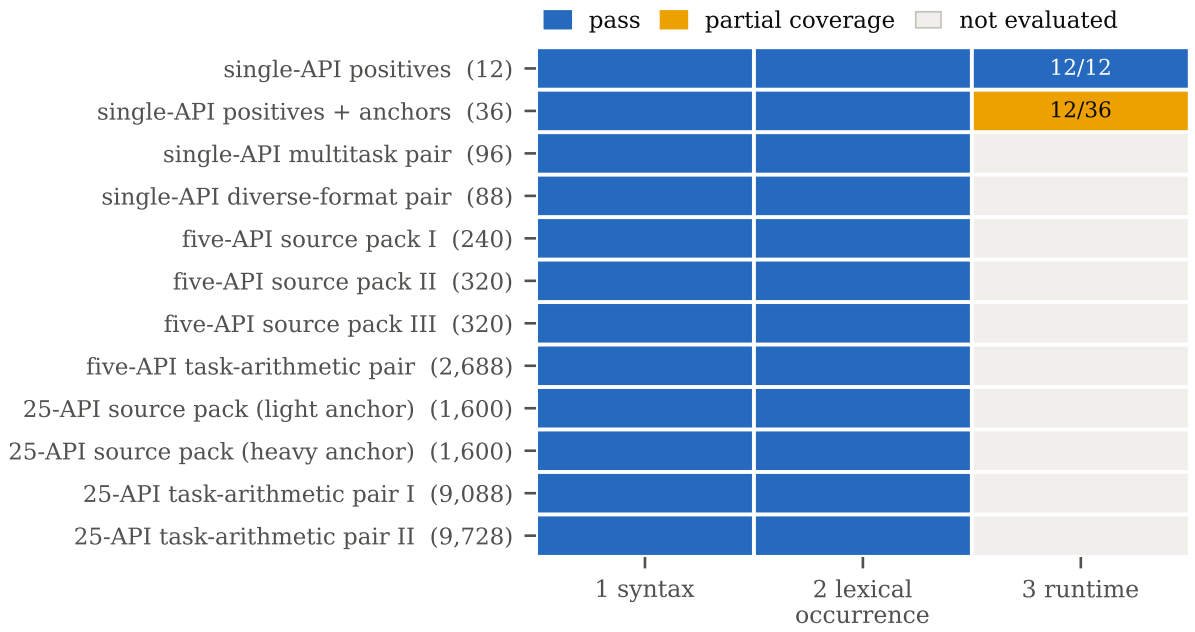


Figure 6.2: Validation status per audited corpus family (rows, with audited row counts) and tier of the three-tier ladder of Section 4.7 (columns). The static tiers, syntax and lexical occurrence, pass everywhere, and runtime evidence covers only the two smallest single-API corpora, which is the certification boundary of the RQ2 answer.

three epoch-expanded task-arithmetic pairs, are repeated training exposures, so corpus size overstates scenario coverage for those inputs.

Each raw five-API or twenty-five-API pack balances target positives one-to-one against hard negatives and, pooled across its APIs, represents all seven registered hard-negative taxa, while the single-API multitask pair covers four output contracts, the diverse-format pair eight task families, and the task-arithmetic pairs three benchmark-surface families. This establishes the declared role, taxon, and task coverage, while it does not measure semantic diversity within a category or independence across corpus versions.

Manual inspection, defined in Section 4.7 as per-example acceptance and per-dataset diversity adequacy, is an author-recorded outcome rather than a machine-audit result. Under the final generation methodology, our review accepted on average 95% of the generated code examples and judged 80% of the generated datasets sufficiently diverse, both rates recorded in our review notes.

Figure 6.2 resolves the same audit by corpus family and validation tier: the static tiers pass for every family, and runtime evidence exists only for the two smallest single-API corpora.

The audit therefore supports a bounded answer to RQ2. The citable data are adequate for the controlled adaptation experiments under the achieved syntax, lexical-occurrence, live source-contract, exact-rendering, complete-pair-overlap, and manual-inspection checks. They are not a corpus-wide

certificate of executable behavior. The next section tests what the smallest audited corpus makes the model produce, while keeping that distinction between data adequacy and model behavior explicit.

6.4. Positive-Only Training and Leakage (RQ3)

Training on the twelve positive target rows moves the target metric immediately (Table 6.5): the best positive-only FIM checkpoint solves all twelve target rows while mentioning the target on nearly three quarters of the non-target rows, broad exact match falls by 18.17 percentage points, and the anachronistic rate rises from 0.36% to 23.71%. The symbol has become producible, but the broad leakage shows that the checkpoint has not learned when the target is appropriate.

The prefix rows of Table 6.5 repeat the failure in the same direction at two intensities: the most aggressive prefix checkpoint (t1) also reaches all twelve target rows at a two-thirds mention rate, while the conservative checkpoint (t0) keeps eleven of the twelve rows with a seventh of that mention leakage and a mildly positive broad change. Positive examples alone make the target available to the decoder, but they do not supply observations in which emitting it is wrong. The result motivates the anchors and matched controls that follow. It does not by itself identify an internal decision boundary or a stored factual representation.

6.5. Anchors and Chat Data Composition (RQ3, RQ4)

The anchored corpora add the observations that positive-only training lacked, examples in which nearby or stable Torch APIs are correct and the target is not, by making 66.7% (24/36) of the corpus anchor rows against 33.3% (12/36) target rows. Table 6.5 summarizes the resulting acquisition and locality frontier across task formats.

Figure 6.3 places every run of Table 6.5, together with the three task-vector arms of Section 6.9, on the joint acquisition and leakage plane, with the mention component of the leakage pair on its horizontal axis. **Anchor FIM** gives the strongest measured acquisition. Its high-recall checkpoint t1 solves all

Table 6.5: Single-API results on the matched 11,485-row broad T1/L2 sample. Target = exact matches on the 12 target rows. ΔEM = broad exact-match change versus baseline in percentage points. Leakage = non-target rows (of 11,473) containing the target string (mention) or resolving to it as the prediction (resolved). † marks the best acquisition-locality tradeoff point of the line and ⋆ the best point in the chat format that the scale-up carries forward, both judged on high target acquisition at low leakage rather than on high broad exact match.

Run	Target	Broad EM	ΔEM (pp)	Leak (mention)	Leak (resolved)
Baseline	0% (0/12)	31.83%	n/a	0% (0/11,473)	0% (0/11,473)
Positive-only FIM (best)	100% (12/12)	13.66%	−18.17	73.19% (8,397/11,473)	50.92% (5,842/11,473)
Positive-only prefix t1	100% (12/12)	15.94%	−15.89	67.65% (7,762/11,473)	42.98% (4,931/11,473)
Positive-only prefix t0	91.7% (11/12)	32.87%	+1.04	9.84% (1,129/11,473)	6.38% (732/11,473)
Anchor FIM t1	100% (12/12)	40.09%	+8.25	13.45% (1,543/11,473)	3.56% (409/11,473)
Anchor FIM t5†	75.0% (9/12)	42.67%	+10.84	0.33% (38/11,473)	0.21% (24/11,473)
Anchor prefix t0	58.3% (7/12)	35.76%	+3.93	0.25% (29/11,473)	0.11% (13/11,473)
Anchor prefix t2	75.0% (9/12)	33.73%	+1.90	0.64% (73/11,473)	0.37% (42/11,473)
Chat, target-heavy (1:1), version comment⋆	75.0% (9/12)	37.88%	+6.05	1.09% (125/11,473)	0.98% (112/11,473)
Chat, balanced (1:2), version comment	58.3% (7/12)	35.08%	+3.25	0.14% (16/11,473)	0.12% (14/11,473)
Staged anchor-only t27	0% (0/12)	34.96%	+3.13	0% (0/11,473)	0% (0/11,473)
Staged target t23	83.3% (10/12)	32.50%	+0.67	3.34% (383/11,473)	2.56% (294/11,473)
Staged target t13	66.7% (8/12)	28.26%	−3.57	9.55% (1,096/11,473)	6.35% (728/11,473)
Staged target t0	66.7% (8/12)	35.23%	+3.40	0.68% (78/11,473)	0.64% (74/11,473)
Staged target t2	50.0% (6/12)	35.12%	+3.29	0.06% (7/11,473)	0.05% (6/11,473)

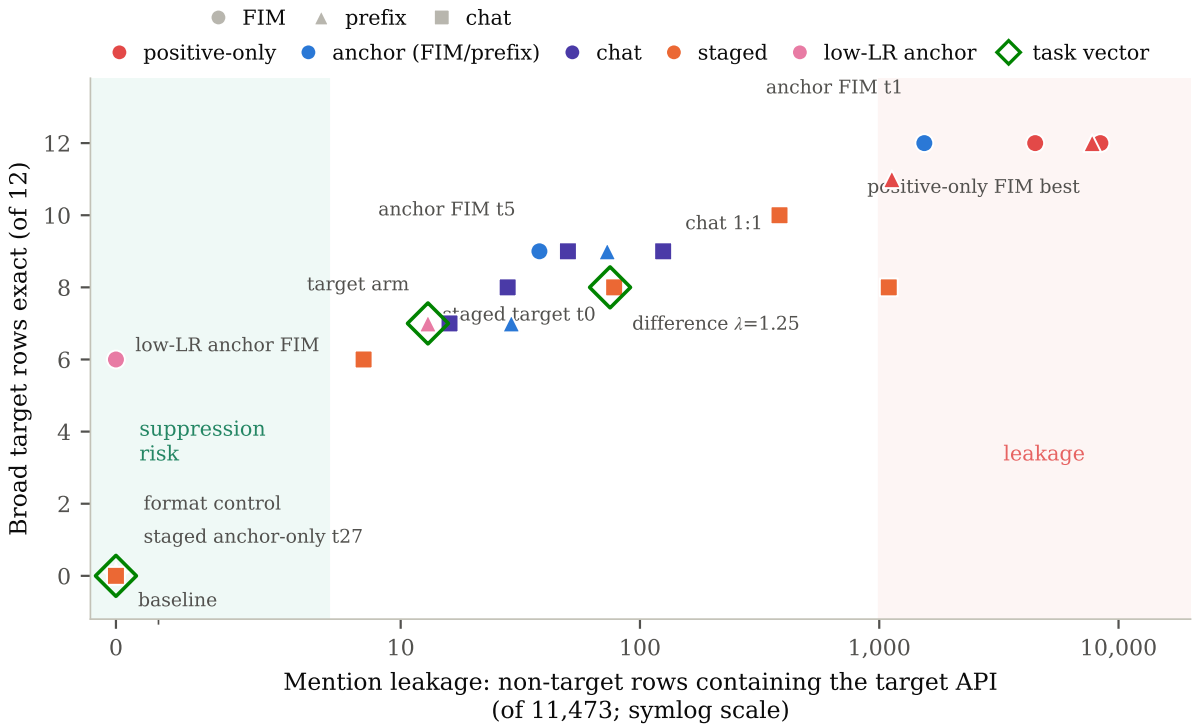


Figure 6.3: Acquisition versus leakage for the single-API SFT runs of Table 6.5 and the three task-vector arms of Section 6.9 on the matched sample. The horizontal axis counts non-target rows (of 11,473) whose completion mentions the target string, and the vertical axis counts exact matches on the 12 broad target rows. Color encodes the data family, marker shape the training task format, and open green diamonds the task-vector arms. Positive-only training reaches the top of the target axis only inside the leakage region, the target-free anchor-only stage and the format control lie at the origin of the suppression region, and the useful adapters, with the difference vector at λ=1.25 among them, occupy the conditional band in between.

twelve broad target rows and the full allowlist (100%, 30/30, Wilson 95% interval 88.6–100.0%) but keeps leaking, with mention leakage that breaks baseline-correct behavior on 2.11% of the off-target rows (242/11,473). Its best tradeoff checkpoint t5 gives up three of t1's twelve target rows and cuts its mention count forty-fold, its residual mention leakage breaks a single baseline-correct row (0.01%, 1/11,473), and it is the best simultaneous acquisition and locality point in the single-API SFT line († in Table 6.5). **Anchor prefix** is more conservative but weaker, reaching at most 75.0% (9/12) at low leakage.

The **chat** experiments use the code-gap instruction format of Chapter 5, where each prompt contains a short instruction, a code snippet with a marked gap, and optionally a version comment, but no documentation passage or target signature to copy. The target-heavy variant with a version-requirement comment is the best chat-format point in Table 6.5, and the balanced (1:2) variant gives up two of its nine target rows together with most of its leakage, so increasing anchor mass within this format and sweep reduces both measured acquisition and leakage. FIM is not a format the thesis can prefer, because its sentinel-token interface is untrained and undocumented for the instruction-tuned assistant model (Section 4.6.2), so the target-heavy chat point is the strongest checkpoint in the format carried into the scale-up (⋆ in Table 6.5).

For RQ4, under the matched sweep budgets of Table 6.5, FIM produces the strongest target acquisition, prefix produces a weaker low-leakage frontier, and chat acquires the target even when the prompt carries no copyable documentation or signature. Across these runs, format and target-to-anchor composition are associated with different acquisition and locality profiles.

6.6. Staged Anchor-Then-Target Training (RQ4)

The staged experiment tests whether the anchor pressure of Section 6.5 must act during target learning, by separating anchor and target exposure in time. An anchor-only stage first produces a checkpoint with 0% (0/12) target acquisition and 0% (0/11,473) mention and resolved leakage, while still improving broad exact match (Table 6.5). Target-only continuation then reaches up to 83.3% (10/12) at checkpoint

t23, but locality worsens along the larger continuations: relative to the initializer, t23 fixes 5.48% of the off-target surface (629/11,473) while breaking 8.03% (921/11,473), a net loss of 2.55% (292/11,473), and raises phantom outputs from 25.43% to 30.91%, while the more aggressive t13 loses a net 6.77% (777/11,473). Checkpoint t0 stays net-positive on the off-target surface (+0.20%, 23/11,473), and the leakage pairs of all staged points appear in Table 6.5.

The conservative checkpoints leak the target almost exclusively onto sibling runtime-state predicates (`torch.jit.is_tracing`, `torch.jit.is_scripting`, `torch.onnx.is_in_onnx_export`), while the aggressive ones broaden into ordinary predicates such as `torch.is_tensor`, so leakage grows outward from the target's semantic neighborhood.

Staged training does not improve on the simultaneous chat frontier: no staged checkpoint dominates the joint profile of the best chat run in Table 6.5, so within these experiments an anchor-trained initializer alone does not reproduce the locality obtained when target and anchor examples remain in the same training stage. The staged trajectory is still diagnostically useful because target recall, leakage, and broad damage change together, making target recall alone an unsafe checkpoint-selection rule.

6.7. Mixed-Task Chat (RQ4)

With the chat format carried forward, rendering the target and anchor examples through four output contracts tests whether target behavior transfers across answer forms: replacement expression, API-call identification, guard-line patching, and semantic JSON. The best mixed-task sweep checkpoint solves 63.3% (19/30, Wilson 95% interval 45.5–78.1%) of the allowlist, matching the optimum of the best single-task chat sweep while spreading supervision over four answer shapes. The fixed-recipe multitask arm later frozen as the task-vector target arm scores four allowlist rows below that sweep optimum, with its full profile in the target-arm row of Table 6.6.

The same arm moves broad non-target exact match by almost four percentage points, and, as the next section shows, a matched target-free control reproduces most of that broad movement, so mixed-task acquisition cannot be interpreted from aggregate broad exact match alone.

6.8. Broad Exact Match and Knowledge-Free Gains (RQ4)

Two runs from Table 6.5 bracket the interpretation problem: the staged anchor-only initializer, trained without target examples, improves broad exact match by +3.13 percentage points, fixing 5.09% of the off-target rows (584/11,473) and breaking 1.96% (225/11,473) with zero target acquisition and zero leakage, while positive-only FIM solves every target row and breaks baseline-correct behavior on 21.08% of the off-target rows (2,419/11,473). A single aggregate score cannot distinguish these cases, the decomposition into target hits, leakage, and paired non-target movement can, and Figure 6.4 shows that the largest broad gains and the strongest target acquisition occur in different runs.

The matched format-control arm, which uses the same task families and step budget but replaces the target with sibling APIs, makes the same point quantitatively: with 0% target acquisition on both target surfaces and 0% leakage, it still changes broad non-target exact match by +3.45 percentage points (5.73% win share, 657/11,473, against a 2.27% loss share, 261/11,473), reproducing all but 0.46 points of the target arm's +3.91. A later ablation control containing no benchmark-formatted rows, no anchors, and no target API still changes broad exact match by +5.45 percentage points. These controls show why aggregate improvement alone is not evidence of target acquisition.

Figure 6.5 adds the outcome-taxonomy view of the same runs, showing which error classes the movement exchanges. The positive-only checkpoints convert correct and no-API mass into wrong-but-valid and anachronistic outputs, and their anachronistic blocks are the taxonomy signature of leakage (Chapter 4): at the positive-only FIM point, 99.7% of the arm's anachronistic rows (2,714/2,723) are the leaked target, and its resolved leaks (50.92% of the 11,473 non-target rows, Table 6.5) split into 3,128 wrong-but-valid rows pinned to torch 2.3 or later and 2,714 anachronistic rows pinned before the target's 2.3.0 introduction. The same signature orders the arms, with the anachronistic share rising from the baseline's 0.36% to 23.71%, 19.97%, and 13.19% across the positive-only FIM best, prefix t1, and FIM t2 checkpoints, staying elevated at the aggressive staged continuations (3.61% at t13, 1.52% at t23) and the high-recall anchor FIM t1 (1.81%), and returning to near-baseline values in every other anchored, staged, and chat arm.

The anchored and chat arms move mass in the opposite direction, drawing down phantom and no-API shares into correct output, although the high-recall anchor FIM checkpoint t1 pushes its phantom

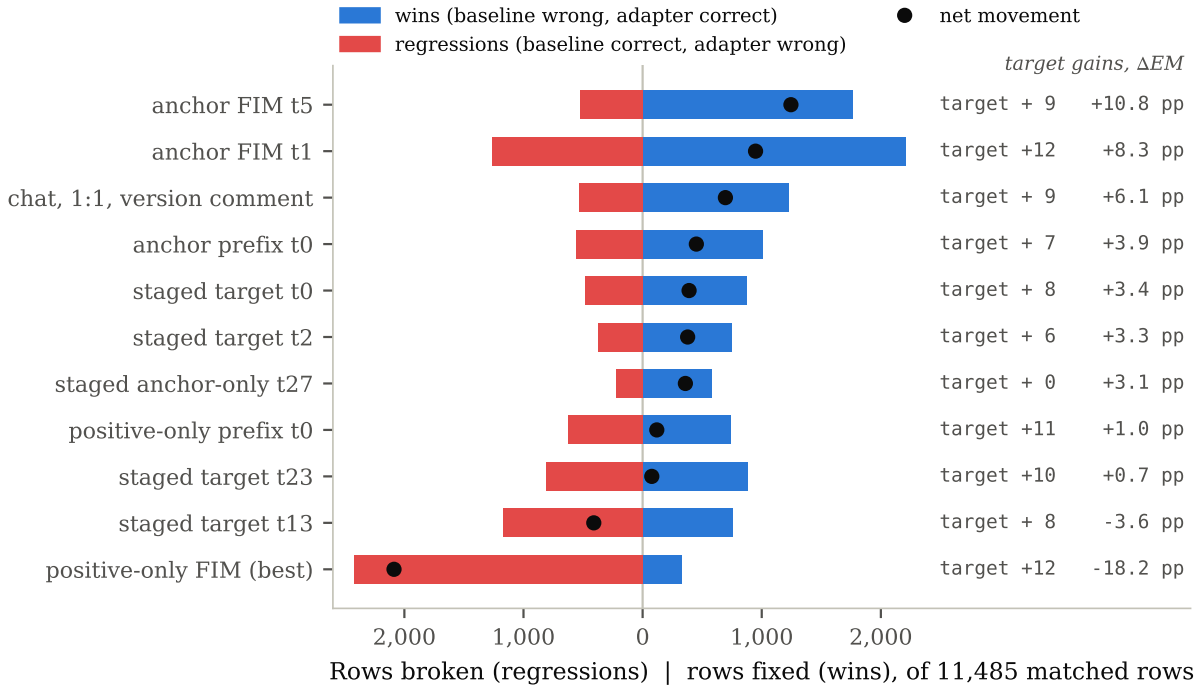


Figure 6.4: Paired row movement versus the baseline on the matched 11,485-row sample, ordered by net movement. Blue bars count rows the adapter fixes, red bars rows it breaks, and the dot marks the net, while the right column gives the target-row gains (of 12) and the broad exact-match change. The target-free anchor-only stage gains +3.1 percentage points with zero target gains, while positive-only FIM gains all 12 target rows and breaks baseline-correct behavior on 21.08% of the 11,473 off-target rows (2,419/11,473).

share slightly above baseline while nearly emptying the no-API class. The target-free anchor-only stage produces a smaller shift of the same anchored shape, so most of this taxonomy movement does not require target examples, and the aggressive staged continuation t23 partially reverses it, regrowing phantom mass beyond its baseline share. At composition level, the target arm and the format control show nearly the same anchored profile, differing most in the no-API share (8.95% against 11.08%), and the difference vector at λ=1.25 keeps a near-baseline anachronistic share (0.58%) and correct share (31.97%) with a slightly raised phantom share (28.71%). Appendix A extends this view to all main runs.

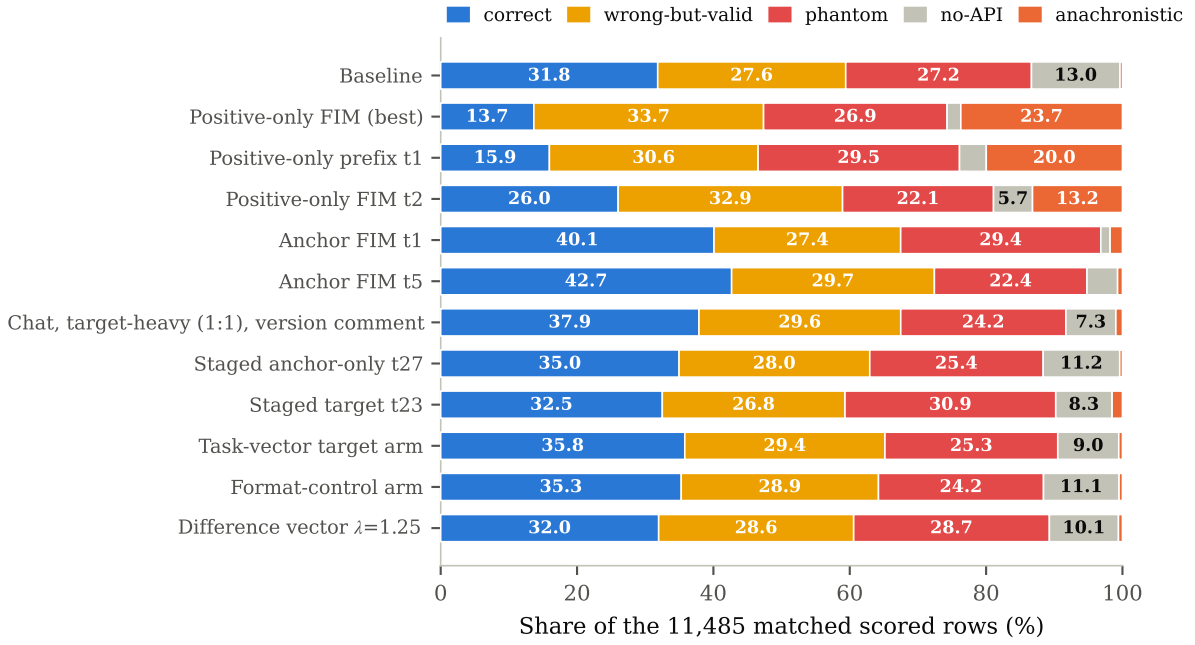


Figure 6.5: Outcome taxonomy on the matched 11,485-row sample for a curated arm subset, with categories as in Chapter 4. The positive-only checkpoints trade correct and no-API mass for anachronistic and wrong-but-valid output, while the anchored, chat, and target-free arms draw down phantom and no-API mass into correct output at a near-baseline anachronistic share. The bottom three bars add the task-vector arms of Section 6.9, whose difference vector keeps the anchored profile at composition level.

6.9. Task Arithmetic (RQ4)

Because the format control reproduces nearly all of the target arm's broad movement (Section 6.8), task arithmetic asks whether that shared task-alignment movement can be subtracted from the target arm in weight space. Table 6.6 reports the reference arms and the λ-sweep of the plain difference vector $K_1 = \Delta_t - \Delta_c$, composed as $\theta = \theta_0 + \lambda K_1$, with this operator and its variants defined in Section 4.5. Figure 6.6 traces the sweep across the three measurement axes.

The main frontier point is the difference vector at $\lambda = 1.25$. Its allowlist and broad-target estimates exceed the frozen target arm's by one row each (Table 6.6), with overlapping Wilson 95% intervals (36.1–69.8% against 33.2–66.8% on the allowlist), so these differences do not resolve an acquisition improvement. Broad collateral changes from +3.91 to +0.07 percentage points, near but just outside the ±0.06 percentage-point evaluation floor, so within this frozen target-control pair the composition retains a comparable target estimate while removing most of the broad movement shared with the

Table 6.6: Task-arithmetic reference arms and the plain-difference λ sweep. Allowlist = exact target matches of 30 diagnostic rows. d_{Coll} = non-target broad exact-match change versus the canonical λ=0 base run (noise floor ±0.06 pp). Leak = mention / resolved rates and counts on 11,473 non-target rows. Churn = gross changed rows versus base.

Model	Allowlist	Broad target	d_{Coll} (pp)	Leak (mention/res.)	Churn
Base θ_0	0% (0/30)	0% (0/12)	0.00	0% / 0% (0/11,473 / 0/11,473)	0% (0/11,473)
Target arm Δ_t	50.0% (15/30)	58.3% (7/12)	+3.91	0.11% / 0.07% (13/11,473 / 8/11,473)	9.23% (1,059/11,473)
Control arm Δ_c	0% (0/30)	0% (0/12)	+3.45	0% / 0% (0/11,473 / 0/11,473)	8.00% (918/11,473)
Locality arm	13.3% (4/30)	0% (0/12)	+4.94	0.01% / 0.01% (1/11,473 / 1/11,473)	11.41% (1,309/11,473)
Difference λ=1.0	0% (0/30)	8.3% (1/12)	+0.14	0.15% / 0.10% (17/11,473 / 11/11,473)	5.47% (628/11,473)
Difference λ=1.25	**53.3% (16/30)**	**66.7% (8/12)**	**+0.07**	**0.65% / 0.46% (75/11,473 / 53/11,473)**	**6.87% (788/11,473)**
Difference λ=1.5	56.7% (17/30)	66.7% (8/12)	−0.26	1.72% / 1.12% (197/11,473 / 129/11,473)	8.04% (922/11,473)
Difference λ=2.0	53.3% (16/30)	66.7% (8/12)	−1.07	5.30% / 3.59% (608/11,473 / 412/11,473)	11.18% (1,283/11,473)

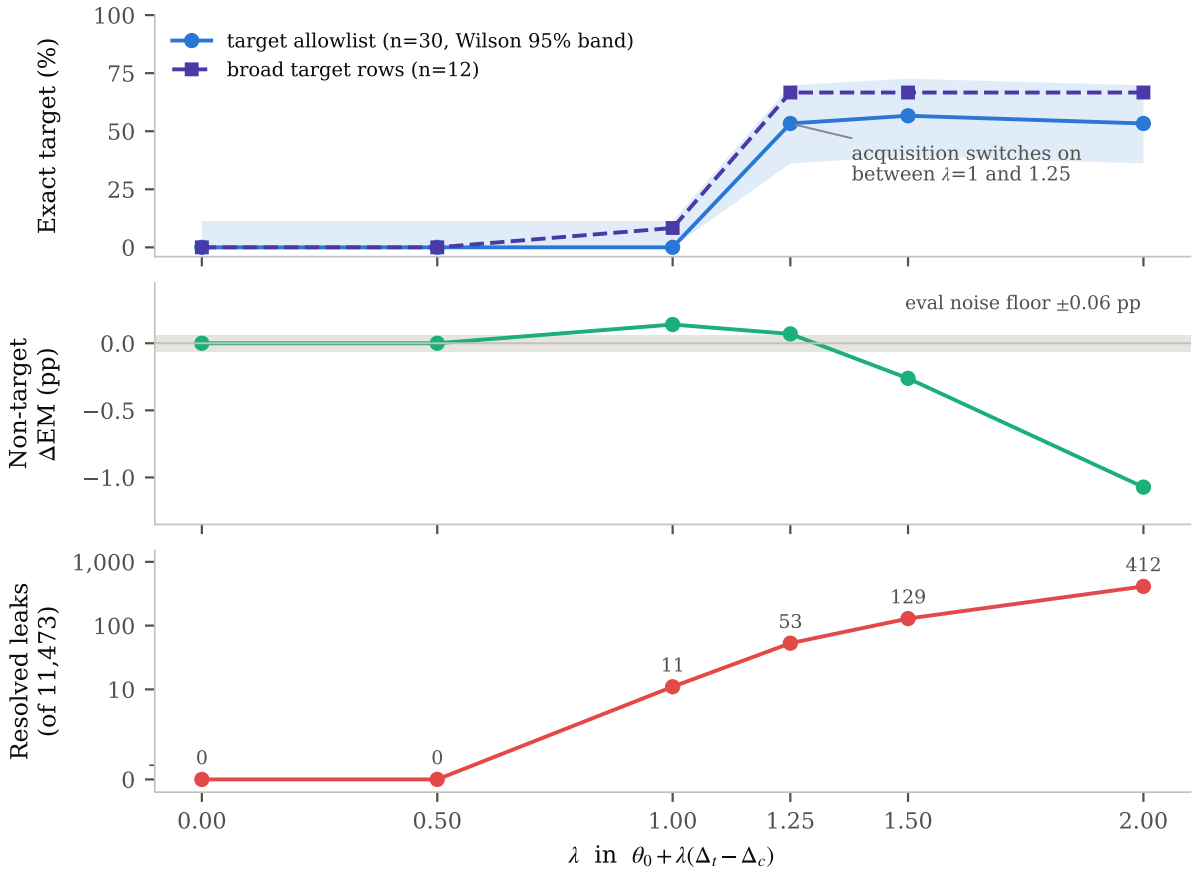


Figure 6.6: The plain-difference λ sweep on its three measurement axes. Top: exact target rates on the 30-row allowlist (with Wilson 95% band) and the 12 broad target rows, where acquisition switches on between $\lambda = 1$ and $\lambda = 1.25$ and then saturates. Middle: non-target broad exact-match movement against the canonical base run, with the ±0.06 percentage-point evaluation noise floor shaded. Bottom: the resolved component of leakage on the 11,473 non-target rows, which grows with λ while acquisition stays flat. Table 6.6 reports the corresponding mention / resolved pairs.

control. Figure 6.3 places the composition among the single-API SFT runs, where it holds eight of the twelve broad target rows at a 75-row mention count, inside the conditional band beside the best chat and staged points and at a near-floor broad collateral that no SFT run of Table 6.5 matches.

Leakage moves in the opposite direction along the sweep (Table 6.6 and Figure 6.6): between $\lambda = 1.25$ and $\lambda = 2$ the resolved leak count grows nearly eightfold while allowlist acquisition gains at most one row, so increasing the difference-vector magnitude past $\lambda = 1.25$ buys little additional acquisition and substantially worsens locality. The tested grid also locates an onset interval, with the allowlist at 0% (0/30, 0.0–11.4%) at $\lambda = 1$ and 53.3% (16/30, 36.1–69.8%) at $\lambda = 1.25$.

All difference-vector claims are *control-relative*. The operator preserves what the target arm contains relative to the selected control, and this control shares task families and sibling-predicate content with the target arm. With a differently constructed control, the same operator removes benchmark elicitation instead of preserving it. Training-run variation at a fixed configuration spans 50.0–70.0% (15–21/30) on the allowlist across three runs, with endpoint Wilson 95% intervals of 33.2–66.8% and 52.1–83.3%, which is as large as many cross-run method gaps, so Table 6.6 is interpreted as a within-pair sweep of one frozen Δ_t and Δ_c. The $\lambda = 1.25$ point also has poor absolute precision, with a measurable leakage pair (Table 6.6) against eight correct broad target rows, so it remains a diagnostic near-floor frontier point rather than a deployment candidate.

6.10. Linear Surgery and Gradient Diagnostics (RQ4)

The operator variants of Section 4.5 test whether a more selective transformation dominates the plain difference for this arm pair, and Table 6.7 with Figure 6.7 shows that none does. The projection and Fisher-weighted operators buy their higher acquisition only by retaining collateral movement,

Table 6.7: Weight-space operator variants and gradient-surgery runs on the frozen single-API arm pair. Allowlist = exact target matches of 30 diagnostic rows with Wilson 95% intervals. d_{Coll} = non-target broad exact-match change versus the canonical base run in percentage points (noise floor ±0.06). The target arm and the plain difference at λ=1.25 repeat from Table 6.6 as references, and the bold row marks the best acquisition-versus-collateral tradeoff, which none of the variants below dominates.

Operator	**Allowlist**	**Wilson 95%**	d_{Coll} **(pp)**
Target arm Δ_t (reference)	50.0% (15/30)	33.2–66.8	+3.91
Plain difference, λ=1.25 (reference)	**53.3% (16/30)**	**36.1–69.8**	**+0.07**
Per-module projection, λ=1.5	76.7% (23/30)	59.1–88.2	+2.54
Rank-truncated projection, r=8, λ=1.5	76.7% (23/30)	59.1–88.2	+2.41
Fisher-weighted subtraction, τ=2	66.7% (20/30)	48.8–80.8	+2.89
TIES-style trimming	36.7% (11/30)	21.9–54.5	−0.21
DARE-style dropping	46.7% (14/30)	30.2–63.9	−0.34
Shared-direction removal, k=1	6.7% (2/30)	1.8–21.3	+2.44
Gradient surgery off (matched run)	70.0% (21/30)	52.1–83.3	+4.54
Gradient projection	53.3% (16/30)	36.1–69.8	+2.48
Anti-leakage terms (two variants)	0% (0/30)	0.0–11.4	−3.98 / −5.12

magnitude sparsification lowers acquisition without improving locality, and removing the top output direction shared with the control collapses acquisition almost entirely. For this frozen pair and the tested operators, useful target behavior is not isolated in a direction that can be removed from the control-shared component without large loss. The operators ran at their registered settings on this one arm pair, without per-operator hyperparameter search, so the comparison bounds what the tested configurations achieve, not what each method could reach with dedicated tuning, and it supports no claim about the geometry of API knowledge beyond this pair.

The online gradient-surgery family gives a compatible result from the training side (Table 6.7): projecting the control direction out of the gradient trades about a quarter of the matched unmodified run's acquisition for roughly half of its collateral movement, while every tested variant with an explicit anti-leakage unlikelihood term reaches 0% (0/30, 0.0–11.4%) and also damages broad behavior. These runs show that the tested penalties did not separate acquisition from non-emission, without proving that no such separation exists, and the leakage instruments are themselves string-based, so part of the observed coupling is definitional.

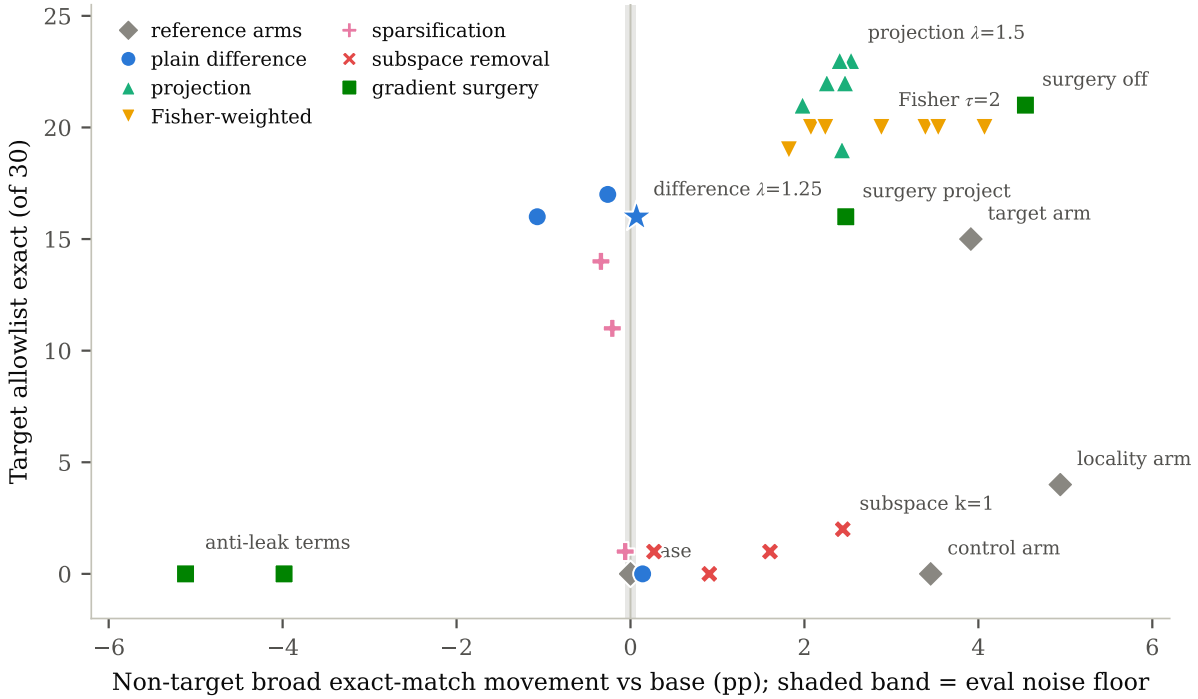


Figure 6.7: All weight-space operator variants on the plane of non-target broad movement (of 11,473 rows, versus the canonical base run) against allowlist acquisition (of 30 rows). Gray diamonds are the reference arms, the star is the plain difference at $\lambda = 1.25$, and the shaded band is the evaluation noise floor. Operators that acquire more than the difference vector do so only by retaining collateral movement, and the variants with explicit anti-leakage pressure destroy acquisition entirely.

6.11. Diverse-Format Data and Benchmark Elicitation (RQ2, RQ4)

Where the arms above vary format within benchmark-shaped completion, the diverse-format v1 target arm is trained without benchmark-style completion templates in its target rows. It reaches only 16.7% (2/12) on the broad target rows and 30.0% (9/30, Wilson 95% interval 16.7–47.9%) on the allowlist. During data development the substring-scored probes elicited the target outside the benchmark format, which motivated the bridge and ablation designs of Chapter 5, and the replicated form of this elicitation gap appears in the five-API lane, where target-string presence remains elevated while exact-match conversion changes across checkpoints and seeds (Section 6.12).

The v2 ablation line provides a larger benchmark-side contrast. A diverse-format arm with bridge rows and no anchor replay reaches 80.0% (24/30, Wilson 95% interval 62.7–90.5%) on the allowlist and 66.7% (8/12) on the broad target rows at +4.71 percentage points of broad movement. Its leakage pair is 0.36% mention (41/11,473) and 0.31% resolved (36/11,473). It is the highest-acquisition single adapter in this experiment line. Its own difference vector, built against a scenario-matched control, lowers allowlist acquisition to 10.0% (3/30, 3.5–25.6%), whereas the multitask pair's subtraction preserves acquisition. The comparison directly shows that the effect of task arithmetic depends on what the selected control shares with the target arm.

Together with the direct audit in Section 6.3, these results show that source-grounded synthetic data can support target behavior across several formats. Benchmark-conditioned elicitation and calibrated non-emission remain separate empirical requirements.

Chat SFT on target positives with anchors and matched-control task arithmetic are the recipes that the scale-up of RQ5 carries forward: the first because chat is the format the instruction-tuned assistant setting supports and anchor mixing is what controls its leakage, the second because its difference vector kept a target-arm-level acquisition estimate at near-floor broad collateral, which no SFT checkpoint matched. The other families each lost on a measured axis, positive-only training to leakage, FIM to its untrained sentinel interface, prefix to its weaker acquisition ceiling, staged training to the simultaneous chat frontier, and the more selective operators to the plain difference.

Table 6.8: The five-API champions of the two scale-up lanes on the 141-row target surface. Acquisition = exact target matches under seed 42. Seeds = three-seed mean and range, characterizing the per-seed peak checkpoints of the frozen recipe for the chat SFT row (the record is that distribution's single-seed maximum) and the pre-registered replication of the fixed composition for the task-arithmetic row. Macro = mean of the five per-API rates. Leak m/r = off-target rows (of 11,436) containing a target string (mention) or resolving to a target (resolved). Leak-broken = baseline-correct off-target rows broken by the union of those two indicators. ΔEM = broad exact-match change versus baseline in percentage points. Churn = off-target rows (excluding leaks) whose outcome flips in either direction.

Operating point	Acquisition	Seeds: mean (range)	Macro	Leak m/r (of 11,436)	Leak-broken	ΔEM (pp)	Churn
SFT record (cosine, step 360)	41.8% (59/141)	peaks 28.4% (17.7–41.8%)	44.9%	1.37% / 1.06% (157 / 121)	0.25% (29)	+10.4	27.2% (3,105)
Task-arithmetic champion (c=0.25, λ=1.0)	76.6% (108/141)	75.2% (73.8–76.6%)	76.2%	1.50% / 1.08% (171 / 123)	0.23% (26)	+5.5	19.3% (2,209)

6.12. Multi-API Scale-Up (RQ5)

RQ5 asks whether the single-API recipe scales to a subset of post-cutoff APIs in one adaptation. Four scale-up lanes are completed: positive-anchor chat SFT and matched-control task arithmetic over the five-API zero-baseline subset of Chapter 5, followed by the pre-specified extension of both recipes to twenty-five APIs. The base model solves 0% (0/141) of the five-API target rows at L2, with the phantom-dominated failure composition reported in Section 6.2.

Both lanes were developed by revising diagnosed failures against the same benchmark, an evaluation-iteration threat that pre-registered criteria, frozen final configurations, and fresh-seed replication mitigate but do not remove.

6.12.1. Positive-Anchor Chat SFT at Five APIs

Along the direct-SFT training trajectories, acquisition and broken baseline-correct rows rise together, and the exact-match peaks vary across seeds: under seeds 42, 43, and 44 the frozen recipe peaks at 41.8% (59/141), 25.5% (36/141), and 17.7% (25/141) of the target rows, a replicated peak mean of 28.4% (40/141). The lane's record, the seed-42 cosine-schedule checkpoint at step 360 with all five APIs nonzero, is therefore the single-seed maximum of that peak distribution rather than the recipe's expected yield, and Table 6.8 reports it next to the task-arithmetic champion of Section 6.12.2.

Figure 6.8 separates target-string presence from exact API match using the diagnostic defined in Section 4.8. At the record, 68.8% (97/141, Wilson 95% interval 60.7–75.9%) of completions contain the expected API leaf or alias, compared with 14.2% (20/141, 9.4–20.9%) at baseline, while 41.8% satisfy the exact scorer. Later checkpoints fall below 13% exact acquisition while target-string presence remains elevated, and two cosine-to-zero runs end at 13.5% (19/141) and 19.9% (28/141) exact acquisition despite target-string presence reaching 66.0% (93/141). This is a measured presence-conversion gap, not evidence of an identified internal knowledge state. Across seeds, peak position also changes from step 360 to 180 to 630, and only 14.9% (21/141) of the target rows are solved at all three peaks, compared with a 44.7% union (63/141).

Outside the record, post-first-epoch checkpoints contain the expected target string on approximately 51.1% (72/141) of the target rows, varying by roughly 7.1 percentage points (10/141) across seeds against the baseline's 14.2%, and the leakage toll at the three seed-specific peaks is nearly fixed, with the pair varying by only eight mention and ten resolved rows on the 11,436 off-target rows (per-seed values in Table A.3). Direct chat SFT reproducibly raises target-string presence, while exact-match conversion is trajectory-sensitive.

6.12.2. Task Arithmetic at Five APIs

The task-arithmetic lane trains one target arm and one format-control arm jointly across all five APIs, defines $K_c = \Delta_t - c\,\Delta_c$, and composes the model on CPU as $\theta = \theta_0 + \lambda K_c$ (Section 4.5). The construction differs from the single-API pair of Section 6.9 in its training surface, its control, and its operator. Both arms are trained on benchmark-surface rows, the fills, replacement expressions, and guard-line patches that the benchmark itself asks for, with receiver-binding rows that tie the mesh method `get_local_rank` to concrete mesh dimensions. The control is route-clean, with no control API sharing an import route with any target. Subtraction is partial, with the control coefficient as an explicit dial operated at $c = 0.25$, $\lambda = 1.0$ in place of the single-API full subtraction at $c = 1$, and the difference of the two rank-16 arms is

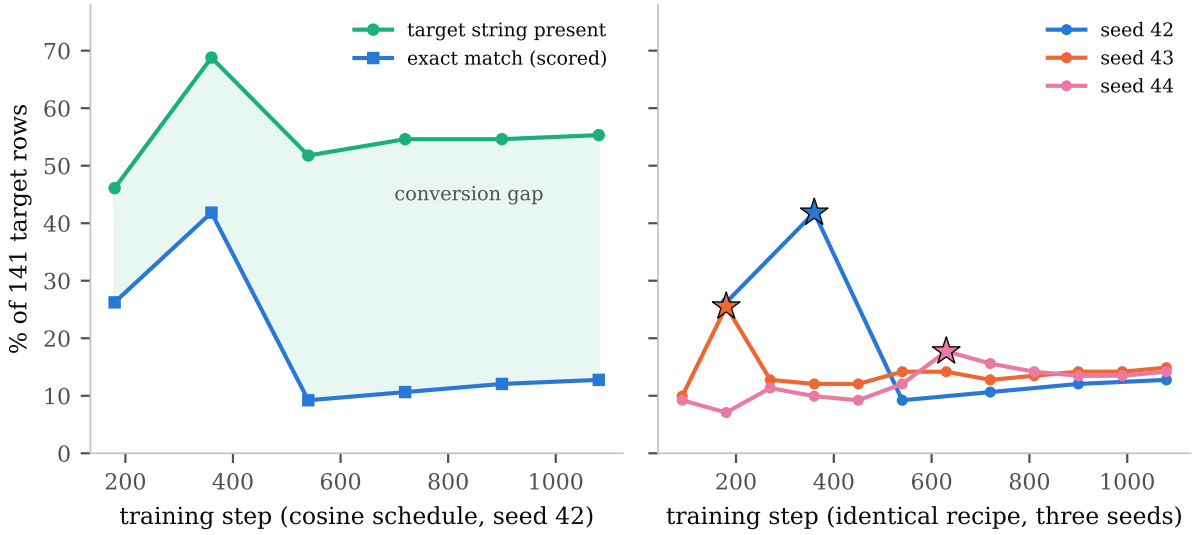


Figure 6.8: Two observations from the SFT scale-up. The left panel compares target-string presence with exact acquisition along the record run, where the exact-match peak at step 360 is transient while the looser presence diagnostic remains elevated. The right panel shows exact-match trajectories under three seeds, with stars marking seed-specific peaks whose height and position vary.

carried as an exact rank-32 vector.

At those coefficients the subtraction lifts the raw target vector's 72.3% (102/141) to the champion's 76.6% (108/141) while lowering the leakage pair from 1.76% mention and 1.37% resolved to 1.50% and 1.08% of the 11,436 off-target rows (Table 6.8). The lane closed with a pre-registered replication in which the champion was rebuilt from arms trained under seeds 43 and 44, with a passing band of ±10 of the 141 rows declared in advance. It passes with a maximum deviation of four rows and a three-seed mean of 75.2% (106.0/141), every API is nonzero at every seed, and the leakage stays equally stable, within 0.10 percentage points on each component of the pair and with 25–27 broken baseline-correct rows across seeds (per-seed values in Table A.2). Acquisition remains uneven across APIs, so the all-nonzero result does not imply uniform performance.

The claim remains control-relative and composition-level: these numbers describe the target arm relative to this format control at the tested coefficients, and task arithmetic does not eliminate leakage, since the champion's pair in Table 6.8 stays measurable at every seed.

6.12.3. Cross-Method Reading and the RQ5 Verdict

Table 6.8 collects the two five-API champions, and Figure 6.9 places all four scale-up checkpoints on the common plane of acquisition against broken baseline-correct rows. The replicated task-arithmetic champion mean of 75.2% stands against a direct-SFT peak distribution whose mean is 28.4% and whose 41.8% maximum is the single-seed envelope, and at seed 42 the champion also produces about half the SFT record's broad exact-match movement (+5.5 against +10.4 percentage points) with less off-target churn (19.3% against 27.2% of the 11,436 off-target rows).

Figure 6.10(a) shows what the two checkpoints do to the target-row failure classes, with per-seed values in Appendix A.5. The composition champion collapses the phantom share from the baseline's 51.1% to 6.4–9.2% across its three seeds while its anachronistic share stays at zero, and at seed 42 correct plus wrong-but-valid predictions cover 89.4% (126/141) of the target surface. The SFT record halves the baseline phantom share to 26.2% at seed 42, and one API is zero at the weakest seed's peak. The receiver-bound `get_local_rank` method and the `custom_op` decorator remain the lowest-acquisition APIs at the task-arithmetic champion, and leakage remains measurable (Table 6.8).

These results answer RQ5 at five APIs affirmatively with a refinement. Both method families produce nonzero acquisition from the zero baseline, but the principal measured tradeoff is acquisition against locality. Direct SFT produces damage-priced checkpoints with variable exact-match peaks, while the matched-control composition replicates within four rows at its fixed coefficients. Retention is evaluated after the twenty-five-API results so that the outside-benchmark cost can be compared across all scales at

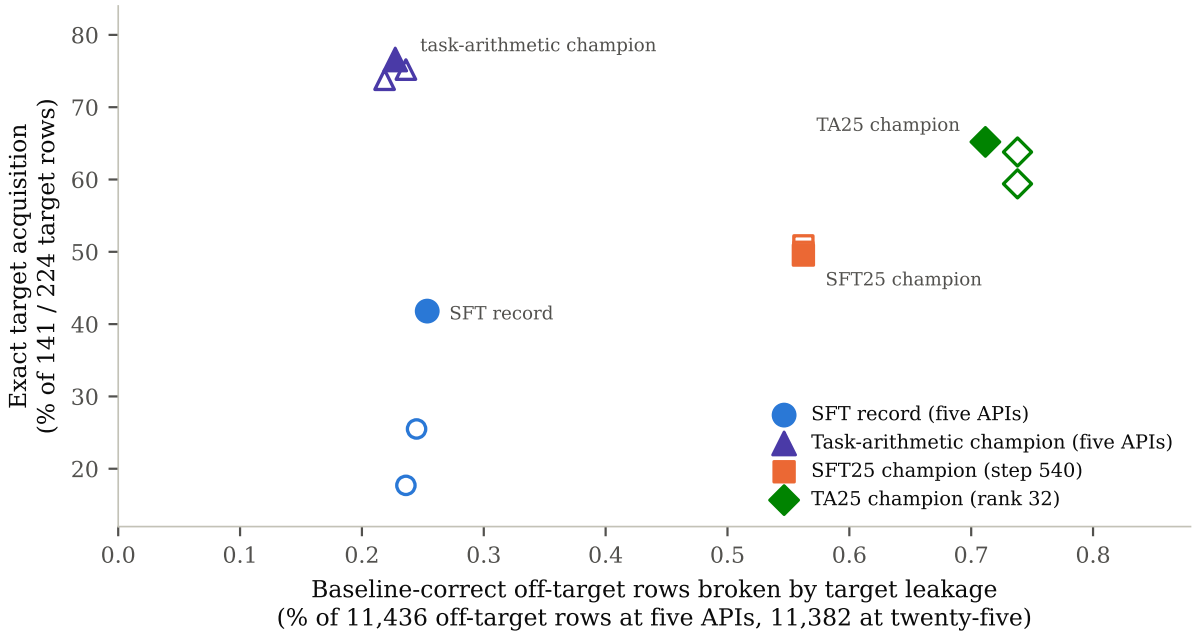


Figure 6.9: The four scale-up checkpoints on the plane of exact target acquisition against baseline-correct off-target rows broken by target leakage, both as percentages of their surfaces (141 target and 11,436 off-target rows at five APIs, 224 asked and 11,382 off-target rows at twenty-five). Filled markers show the seed-42 checkpoints and open markers the replication observations, where the SFT record's replication points are the other two seed-specific peaks and the SFT25 champion's are its step-matched seed scores, drawn at the seed-42 leakage coordinate because that replication is scored on the target surface.

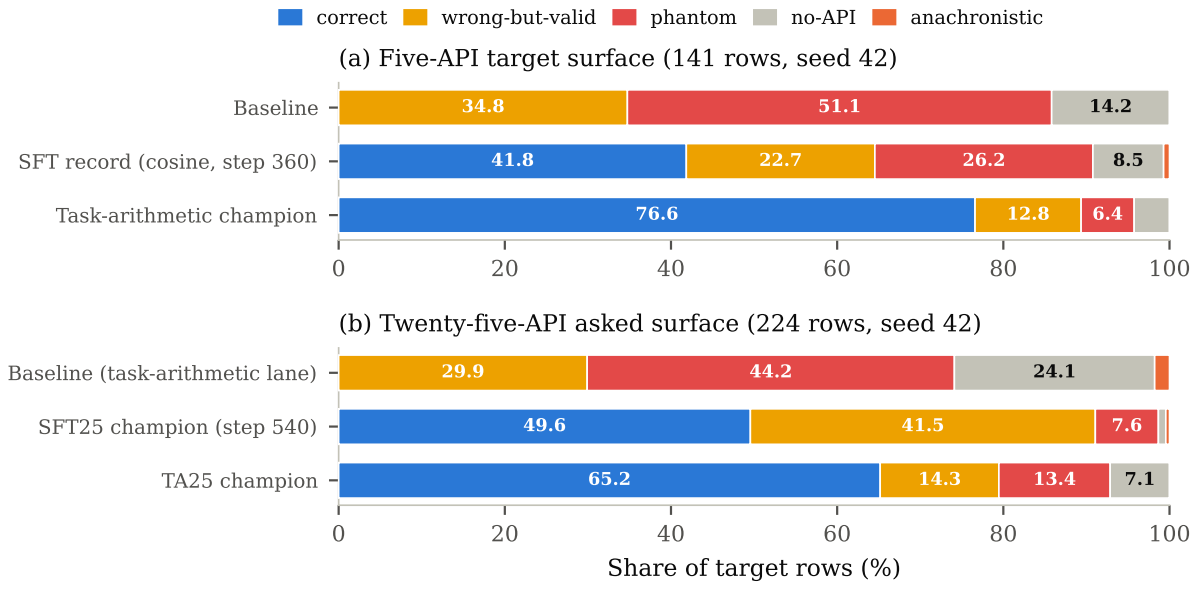


Figure 6.10: Outcome taxonomy on the scale-up target surfaces, with the classes of Chapter 4. (a) The 141-row five-API target surface under seed 42: baseline, the chat SFT record, and the task-arithmetic champion. (b) The 224-row asked surface of the twenty-five-API set under seed 42: baseline, the SFT25 champion, and the TA25 champion. The twenty-five-API baseline bar is the task-arithmetic lane's served baseline, and the SFT lane's independently served baseline agrees with it within one row per class. Figure A.3 extends this view with per-seed values.

Table 6.9: The twenty-five-API champions of the two lanes, reported percentage first. Acquisition = exact matches on the 224 asked rows under seed 42. Seeds = replicated mean and range, over step-matched seeds 43 and 44 for the SFT lane and rebuilt compositions for the task-arithmetic lane. New = the 83 asked new-API rows. Legacy = the 141 validated-five rows. Leak m/r = off-target rows (of 11,382) containing a target string (mention) or resolving to a target (resolved). Leak-broken = baseline-correct off-target rows broken by the union of those two indicators. ΔEM = broad exact-match change versus the lane's baseline serve in percentage points. Churn = off-target rows (excluding leaks) whose outcome flips in either direction.

Operating point	Acq. (224)	Seeds: mean (range)	New (83)	Legacy (141)	Leak m/r (of 11,382)	Leak-broken	ΔEM (pp)	Churn
SFT25 champion	49.6%	50.1%	47.0%	51.1%	2.75% / 2.81%	0.56%		29.6%
(damage-priced, step 540)	(111/224)	(49.6–50.9%)	(39/83)	(72/141)	(313 / 320)	(64)	+8.0	(3,371)
TA25 champion	65.2%	62.8%	48.2%	75.2%	3.12% / 3.00%	0.71%		28.1%
(r=32, c=0.25, λ=1.0)	(146/224)	(59.4–65.2%)	(40/83)	(106/141)	(355 / 342)	(81)	+10.1	(3,201)

once.

6.12.4. Scaling to Twenty-Five APIs

Both lanes were then extended to the frozen 25-API target set of Chapter 5, applying the frozen five-API recipes to twenty additional zero-baseline APIs without per-API tuning. Acquisition uses the asked-at-L2 denominators defined there: 224 rows, of which 141 belong to the validated five APIs (the legacy cohort) and 83 to the fourteen new APIs the benchmark queries at L2 (the new cohort), while the six new APIs without L2 rows are excluded from the L2 micro-averages. The base model reaches 0% (0/224) on the asked surface, with the failure composition reported in Section 6.2. Table 6.9 reports the two champions, with per-API detail in Table 5.5 and per-seed detail in Appendix A.4.

The direct-SFT lane trains one rank-64 adapter on the 25-API pack family, which renders 1,600 audited source rows into 14,400 training rows, and trains a five-API interference control beside it. The registered damage-priced rule selects step 540, the SFT25 champion, whose step-matched seeds land within three rows of one another and whose replicated mean of 50.1% (112.3/224) is the headline SFT result (Table 6.9). Higher-acquisition checkpoints exist along the run but carry more measured damage under the selection score.

At the selected checkpoint, 85.7% (12/14) of the L2-evaluable new APIs are nonzero. The six L2-excluded APIs are evaluated separately on the code-only L0 surface, where step 540 answers 73.1% (49/67) of their rows, a result that cannot be combined with the 224-row L2 micro-average (Appendix A.6). The legacy-interference comparison, taken at the run's highest-acquisition checkpoint, puts the legacy cohort 14.2 percentage points below the paired five-API interference control (53.2% against 67.4% of the 141 rows), inside the pre-registered 15-point bar, and the effect is uneven across APIs, so the aggregate pass does not imply uniform preservation.

The task-arithmetic lane applies the frozen matched-control recipe to the twenty new APIs on the same benchmark instrument, under the criteria registered before launch. The replicated TA25 champion mean of 62.8% (140.7/224) splits into a new-cohort mean of 47.8% (39.7/83) and a legacy-cohort mean of 71.6% (101.0/141), which is 3.5 percentage points (5/141) below the five-API replication's 75.2% and inside the same interference bar.

Per-API replication is assessed only for the fourteen new APIs with L2 rows, of which 71.4% (10/14) have exactly the same acquired-row count across the three champion seeds and 92.9% (13/14) vary by at most one row, while the six new APIs without L2 observations are excluded from this invariance denominator. The rank-32 capacity arm exceeds rank 16 by 3.6 percentage points on the evaluable new-API micro-average, below the pre-registered 8-point criterion. These are measurements on the asked surface only.

On replicated means, the TA25 champion reaches 62.8% acquisition against 50.1% for the SFT25 champion, and it buys those 12.7 percentage points with a somewhat larger seed-42 footprint, 0.71% against 0.56% of the off-target rows broken and a slightly larger leakage pair (both pairs in Table 6.9).

Figure 6.10(b) gives the corresponding failure-class view on the asked surface. The TA25 champion cuts the phantom share from the baseline's 44.2% to 13.4–16.5% across seeds, again with zero anachronistic rows, and the SFT25 champion reaches a lower phantom share still, 7.6% (17/224), while leaving most residual failures as wrong-but-valid choices (41.5%, 93/224), so correct plus wrong-but-valid predictions cover 91.1% of its asked rows against the TA25 champion's 79.5% at seed 42, a difference in residual failure composition that these measurements do not attribute to a cause. On the broad

surface, chat SFT nearly empties the no-API class, from 12.67% of the 11,485-row sample at the scale-up baseline serve to 1.01% at the SFT25 champion. The broad anachronistic class tracks leakage, reaching 1.72% (198/11,485, 133 rows resolving to a leaked target) at the SFT25 champion against 1.51% (173, 122 leaked) at the TA25 champion, while the baseline serve resolves to none of the twenty-five targets on any off-target row. The 25-API study is an intermediate same-instrument stress test and does not establish release-scale or out-of-instrument generalization.

6.13. Retention on Control Benchmarks

The retention suite measures behavior outside the API benchmark. The official scores in Table 6.10 serve only as protocol sanity anchors confirming that the local harness produces plausible base-model scores, with their published sources and harness-specific gaps documented in Chapter 5, Section 5.2, and every adapter comparison uses the identically evaluated local base model and the same item-level scoring path. The six pairable lanes are MMLU, MMLU-Pro, GSM8K, IFEval, HumanEval+, and MBPP+, tested with the exact conditional-binomial McNemar and Holm procedure of Chapter 4 within the paired families fixed in Chapter 5, while DS-1000 is aggregate-only and enters no paired inference.

No single-API cell survives correction, and the family-minimum exact p, given in the caption of Table 6.10, occurs at the difference vector's MBPP+ cell, the largest paired code delta in the table, whose Holm-adjusted value is still 1.0. The chat arm's aggregate GSM8K delta is exactly zero despite equal win and loss shares of 3.03% (40/1,319 each), illustrating why paired churn is reported alongside the point estimate.

All four IFEval point estimates are negative, from −1.11 to −2.22 percentage points (Figure 6.11), but none is significant, the serial-client results are bit-exact, and concurrent-client repeats, which varied by up to 2.2 percentage points on identical weights, stay out of inference, so the common sign is a follow-up signal rather than evidence of a shared instruction-following cost. The suite also shows how easily a retention measurement goes wrong: a superseded configuration that prompted the instruct model with raw code completion instead of its chat protocol manufactured a nominally significant code regression that the canonical protocol shows does not exist, and nothing about the erroneous configuration announced itself, since the runs completed and the paired statistics were internally consistent. Only the failure to reproduce published base-model numbers exposed the problem, which is why every retention claim in this thesis is a paired delta within a protocol first checked against published values.

6.13.1. Retention of the Scale-Up Champions

The four five-API operating points and the three twenty-five-API operating points pass the target-score verification gate before retention evaluation. Table 6.11 and Figure 6.12 report the five-API comparisons, while Table 6.12 reports the twenty-five-API comparisons under the same six pairable lanes and local base outputs.

No five-API cell survives correction, and the family-minimum exact p occurs for the low-leak SFT arm on IFEval (caption of Table 6.11). Both SFT rows have negative MBPP+ point estimates, but because no cell survives the family correction and the arms share evaluation items, this is a sub-threshold pattern rather than a finding. The task-arithmetic champion has no significant paired delta and its largest absolute point estimate is 1.48 percentage points. This does not prove that composition preserves every unmeasured capability.

Table 6.10: Single-API retention deltas versus the locally evaluated base model in percentage points, with wins/losses counting newly solved and newly failed items. Across 24 paired tests, the minimum exact McNemar p is 0.076813 and the minimum Holm-adjusted p is 1.0. Official values are sanity anchors, not effect baselines. † IFEval uses deterministic serial-client runs. ‡ DS-1000 is aggregate-only.

Benchmark	n	Base	Official	Chat 1:1 +ver.	Staged t0	Multitask t13	Diff. λ=1.25
MMLU	14,042	74.24%	74.2	+0.02 (78/75)	+0.06 (59/51)	+0.01 (50/49)	+0.01 (54/52)
MMLU-Pro	12,032	57.50%	56.3	+0.02 (860/858)	+0.02 (801/799)	−0.07 (795/803)	−0.26 (677/708)
GSM8K	1,319	86.35%	91.6	0.00 (40/40)	−0.15 (27/29)	+0.53 (30/23)	−0.53 (17/24)
IFEval†	541	71.53%	71.2	−2.22 (24/36)	−1.11 (18/24)	−1.48 (20/28)	−1.48 (20/28)
HumanEval+	164	76.22%	≈ 76	−1.22 (5/7)	+0.61 (6/5)	−0.61 (3/4)	+0.61 (3/2)
MBPP+	378	70.37%	≈ 69	−0.53 (9/11)	−1.06 (10/14)	+0.79 (7/4)	−2.12 (4/12)
DS-1000‡	1,000	28.20%	n/a	+0.60	+0.10	+0.10	−0.20

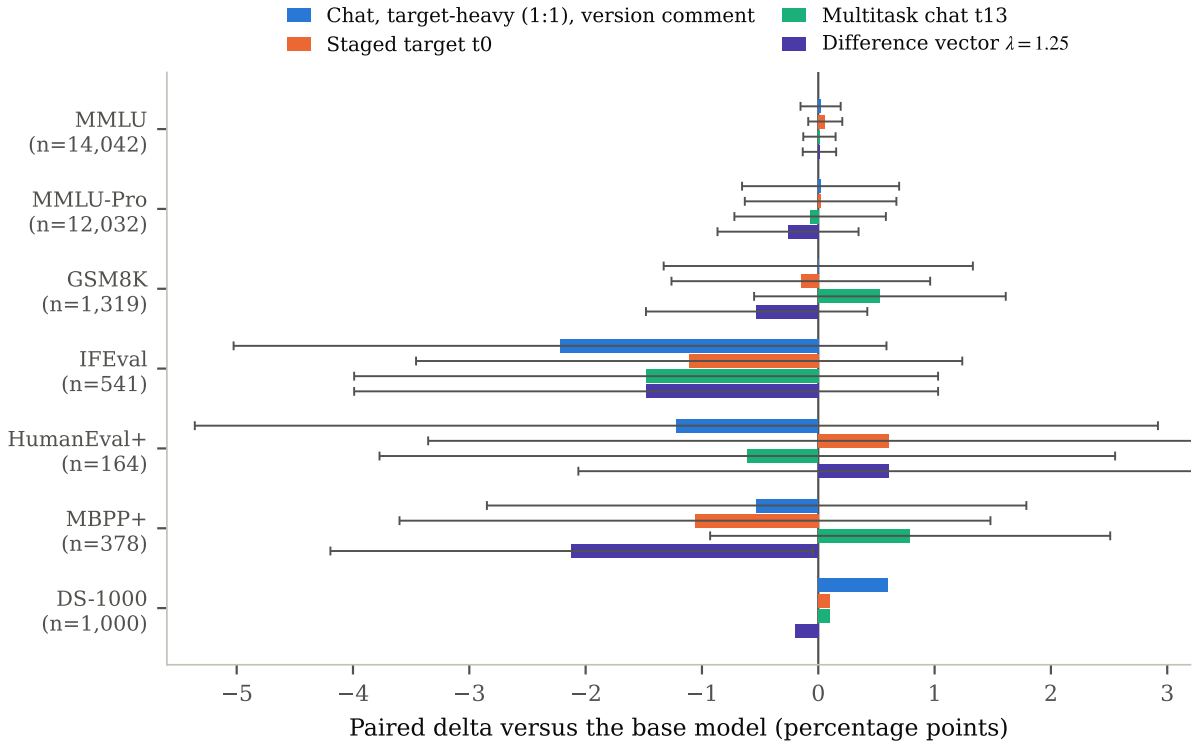


Figure 6.11: Paired retention deltas versus the local base model for the four single-API recipe champions. Whiskers are descriptive normal-approximation 95% intervals computed from discordant-pair counts. Inference uses the exact conditional-binomial McNemar tests and Holm correction reported in Table 6.10. DS-1000 is aggregate-only and is drawn without whiskers, while IFEval uses deterministic serial-client runs.

Table 6.11: Five-API retention deltas versus the local base model in percentage points, with wins/losses. Across 24 paired tests, the minimum exact McNemar p is 0.04139143 and the minimum Holm-adjusted p is 0.99339429. † IFEval uses deterministic serial-client runs. ‡ DS-1000 is aggregate-only.

Benchmark	n	SFT record	SFT low-leak	Task-arithmetic champion	Task-arithmetic knee
MMLU	14,042	−0.05 (227/234)	+0.12 (257/240)	+0.05 (97/90)	−0.17 (71/95)
MMLU-Pro	12,032	−0.14 (1042/1059)	−0.64 (1023/1100)	+0.13 (948/932)	−0.36 (893/936)
GSM8K	1,319	−0.30 (47/51)	−0.83 (48/59)	−0.53 (31/38)	+0.23 (34/31)
IFEval†	541	−2.03 (34/45)	−3.33 (26/44)	−1.48 (25/33)	−2.22 (27/39)
HumanEval+	164	0.00 (7/7)	−1.83 (9/12)	+0.61 (5/4)	+1.22 (5/3)
MBPP+	378	−3.44 (14/27)	−3.70 (14/28)	−0.26 (12/13)	−0.79 (12/15)
DS-1000‡	1,000	+2.30	−0.20	+0.80	+0.70

Table 6.12: Twenty-five-API retention deltas versus the local base model in percentage points, with wins/losses. In the six-test SFT family, GSM8K has exact $p = 0.00066672$ and Holm-adjusted $p = 0.00400034$. IFEval is nominal only. Across the twelve paired tests for the two task-arithmetic points, the minimum exact p is 0.45825832 and the minimum Holm-adjusted value is 1.0. † IFEval uses deterministic serial-client runs. ‡ DS-1000 is aggregate-only.

Benchmark	n	SFT25 champion (step 540)	TA25 champion	TA25 knee
MMLU	14,042	+0.32 (295/250)	−0.06 (188/197)	−0.08 (135/146)
MMLU-Pro	12,032	−0.51 (1067/1128)	−0.02 (1035/1038)	+0.04 (980/975)
GSM8K	1,319	**−2.88** (41/79)	+0.45 (50/44)	−0.23 (44/47)
IFEval†	541	−3.88 (30/51)	−1.11 (31/37)	−1.11 (30/36)
HumanEval+	164	−3.66 (8/14)	0.00 (5/5)	+0.61 (9/8)
MBPP+	378	−0.26 (15/16)	−1.32 (12/17)	−0.79 (11/14)
DS-1000‡	1,000	+0.40	+1.30	+1.30

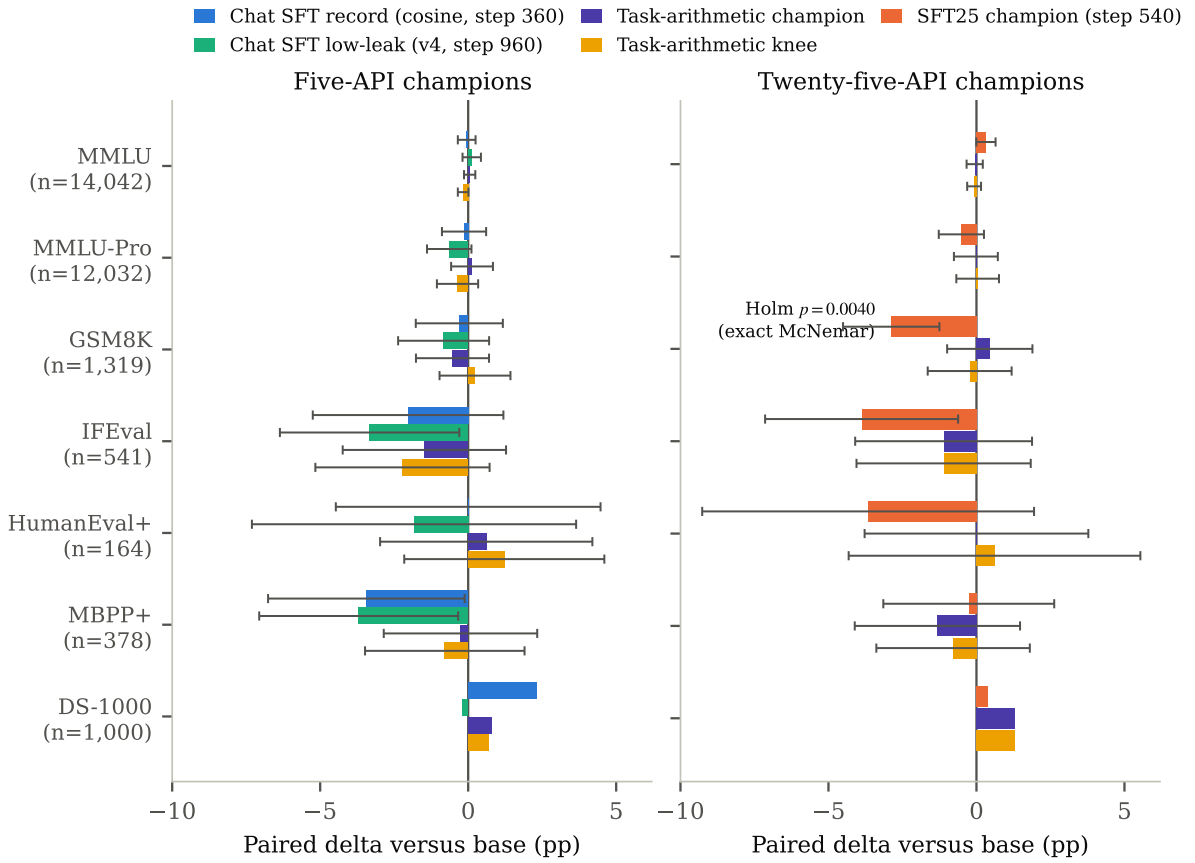


Figure 6.12: Paired retention deltas versus the local base model for the four five-API arms (left) and the three twenty-five-API arms (right). Whiskers are descriptive normal-approximation 95% intervals computed from discordant-pair counts. Inference uses exact conditional-binomial McNemar tests with Holm correction as reported in Tables 6.11 and 6.12. DS-1000 is aggregate-only and is drawn without whiskers. IFEval uses deterministic serial-client runs. The marked GSM8K cell is the only Holm-significant retention regression.

The SFT25 champion is the only arm of the programme with a corrected retention signal, a GSM8K decline significant within its six-test family, in which newly failed items outnumber newly solved ones roughly two to one (79 against 41 of 1,319, Table 6.12). Its IFEval delta is nominal only, the other four pairable SFT lanes do not survive correction, and the DS-1000 aggregate rises without an item-level test.

Both 25-API task-arithmetic points are null across their twelve-test family (both minima in the caption of Table 6.12), and the champion's GSM8K point estimate is positive. This cross-method contrast localizes the observed regression to the tested SFT package but does not identify its cause because method, data, objective, adapter rank, and update construction differ together, and each retention arm has one protocol-standard evaluation seed. The evidence therefore does not attribute the regression to target count, edit mass, or any single recipe component.

6.14. Research-Question Synthesis

RQ1. Benchmark-wide, the base model's errors fall mainly into wrong-but-valid substitutions of semantic siblings or deprecated ancestors of the gold API and phantom names that truncate real module paths, while anachronistic output is nearly absent (Section 6.2). On the post-cutoff targets the model reaches zero exact matches on every frame, 0/12 broad and 0/30 allowlist for the controlled API and zero again at five and at twenty-five APIs, so the failure is structured on the tested API-identity surfaces and invisible to aggregate exact match alone. Signature correctness was not an evaluation surface.

RQ2. The audited corpora pass their static checks completely, syntax parsing and expected-API occurrence, with zero complete prompt-answer overlap against the full T1 benchmark in both audit

views, passing live source contracts, and the author-recorded manual-inspection rates reported in Section 6.3, while 62.58% of structural rows are planned epoch repeats and mechanical runtime and semantic-diversity evidence remains incomplete. The data are adequate for the controlled experiments under the named checks, and certified no further.

RQ3. LoRA SFT makes the missing API producible, positive-only training leaks it across non-target rows (73.19% mention leakage at 100% target recall), and the best anchored point reaches 75.0% (9/12) while its residual mention leakage breaks a single baseline-correct row (Section 6.5), conditional target use on the measured benchmark rather than universal API competence.

RQ4. Format, data composition, and control choice set the measured frontier: a target-free control reproduces +3.45 of the target arm's +3.91 percentage points of broad movement with zero target acquisition, and the $\lambda = 1.25$ difference vector holds a comparable target estimate at a near-floor +0.07 percentage points of broad collateral, a control-relative, pair-specific result that establishes no universal separable subspace (Sections 6.8–6.11). The 24-test single-API retention family is Holm-null.

RQ5. At five APIs the replicated task-arithmetic champion reaches 75.2% against direct SFT's 28.4% peak mean and 41.8% single-seed envelope, and at twenty-five APIs the asked-surface means are 62.8% for the TA25 champion and 50.1% for the damage-priced SFT25 champion, all on the same instrument rather than held-out or release-scale generalization (Section 6.12). The five-API retention family is Holm-null, while the twenty-five-API SFT package carries the programme's only corrected significant regression, −2.88 percentage points on GSM8K, which both task-arithmetic points lack, a package-level contrast without an identified causal component (Section 6.13.1).

7

Discussion

7.1. Interpretation of Findings

Post-cutoff API acquisition turned out to be the cheap half of the problem: twelve source-grounded positives and a rank-16 LoRA adapter make a 7B model produce a symbol it had never emitted. Positive examples cannot teach the condition on that use, because every one of them shows a context where the target belongs and none shows a context where it must not appear. The collapsed checkpoints show the missing condition directly, since they emit the leaked target indifferently on rows whose pinned version predates the API's existence, and the anachronistic class rises and falls with leakage across arms (Section 6.8). The arms that restrained emission were trained on counterexamples, and under the matched FIM sweep budgets the anchored corpus cut mention leakage from 73.19% to 0.33% of the off-target rows while giving up three of the twelve target rows.

The after-the-fact separations recovered less than those data-side counterexamples: gradient projection traded about a quarter of the matched run's acquisition for half of its collateral movement, and both explicit anti-leakage penalties lost acquisition entirely. The pattern is consistent with a limit of supervised fine-tuning itself, whose objective rewards producing the target on the training rows and discourages wrong-context emission only through further examples, and whether reward-level pressure can price leakage where these token-level penalties failed is the reinforcement-learning study specified later in this chapter.

Beside the leakage limit, the experiments exposed an elicitation limit: the diverse-format arm produced the target on its development probes while converting only 16.7% (2/12) of the broad target rows, and along the λ sweep the movement shared with the format control subtracts away at near-floor collateral while resolved leakage grows nearly eightfold past λ=1.25 for at most one more allowlist row. In the adapted model, knowledge and its elicitation are intertwined rather than separable components, in line with the probing literature, where what a model appears to know depends on the prompt used to elicit it and any single prompt measures only a lower bound [66], [67]. The benchmark's own evaluation finds the prompt-level form of the same asymmetry, with in-context documentation raising API accuracy by 10 to 20 points while an explicit version constraint produces no measurable improvement [25]. From training dynamics, Gekhman et al. find that fine-tuning mainly teaches a model to surface what it already knows and that hallucination against pre-existing knowledge rises as new facts are eventually learned [32]. Task arithmetic reports near-orthogonal task vectors, speculating that this orthogonality enables their low-interference addition, and observes the orthogonality decay for semantically similar tasks, the case of same-library APIs [48], while the merging literature documents the resulting parameter interference and builds its procedures around resolving it [49].

7.2. Implications for Library-Version Specialists

The single-API adapter is a diagnostic probe, and a specialist needs a broader contract. Its target set must come from a versioned public surface, because phantom classification, anachronistic classification, and hard-negative construction all depend on availability evidence, and its training data must include shared negative space among taught APIs, especially for siblings that compete for the same receiver

or namespace. Selection must then operate per API and weigh mention leakage, resolved leakage, baseline-correct regressions, churn, and retention alongside acquisition.

On this instrument and at these scales the composition holds the stronger position, since at twenty-five APIs the replicated composition champion exceeds the damage-priced SFT selection by 12.7 percentage points of mean acquisition while breaking baseline-correct completions on a comparable share of the off-target rows (0.71% against 0.56% under seed 42), though with a somewhat larger leakage pair. Both composition points also carry a null retention profile while the SFT package carries the programme's only corrected regression, so a builder starting from these results would start there. The preference stays bounded to the tested packages, because the retention contrast compares bundles that differ in data, objective, rank, and update construction, and a production choice would still need replicated acquisition and retention under the intended data, latency, and serving constraints.

The corpus quality that RQ2 certifies rests partly on manual inspection, the per-example acceptance and per-dataset diversity reading defined in Chapter 4. We reached the final generation methodology only after revising earlier teacher prompts until quality and diversity were acceptable, and the review reads every generated dataset, so the cost of that gate grows with every corpus and every revision round. It also does not scale, because a release-scale specialist would need data for hundreds of APIs (the strict benchmark matrix behind the twenty-five-API selection alone contains 1,151 introduced names, 146 of them with at least one benchmark row), so automated quality and diversity validation would have to replace author reading. Chapter 8 takes that replacement up as future work, LLM agents that investigate the API they teach and validate their own examples.

The twenty-five-API study also remains transfer within one instrument: the recipes were never tuned on the twenty new APIs, but the questions still come from the LibEvoBench task family [25], and full release coverage would need a larger target pool with benchmark coverage for the skipped APIs together with evidence across libraries and model families. A deployed specialist would further need runtime version selection, behavior-level validation of generated code, rollback criteria, and monitoring of newly taught targets in unsupported contexts, none of which is tested here.

7.3. Why Reinforcement Learning Was Not Part of the Final Evidence

We considered reinforcement-learning (RL) post-training through GRPO- and DAPO-style group optimization [36], [37], [38] and excluded it from the completed evidence, because the central ambiguity in every experiment was never how to maximize the target metric, positive-only SFT already saturating target recall, but what the model actually learned and what else moved, and a policy-optimization stage added before those measurements stabilized would have layered reward-design and optimization confounds on top of the same identification problem. The natural naive reward, exact target match, is also the objective whose supervised analogue produced the target-string prior, so group-based RL on that reward would plausibly amplify leakage, and the gradient-surgery results suggest a leakage penalty is not trivially stabilizable, since explicit anti-emission pressure destroyed acquisition in the supervised setting. The thesis therefore contains no empirical conclusion about RL effectiveness for versioned APIs.

An RL study can now be specified around the measured failure modes. Rollout batches would need target contexts, anchors, and hard negatives, while rewards would combine exact target identity, version validity, and completion-format validity with penalties for resolved target leakage, phantom paths, and regressions on baseline-correct controls. Full-string similarity alone cannot enforce API identity. The falsifiable comparison is whether locality-aware RL improves the acquisition-locality frontier of a fixed SFT input adapter relative to a reward-ablated control under the same paired movement, churn, and retention measurements. Until that experiment is run, RL remains scoped future work.

7.4. Limitations and Threats to Validity

Construct validity. Exact dotted-path match measures whether the required API identity appears, while it does not establish full program correctness. Mention leakage is raw string containment and resolved leakage depends on the prediction extractor, so both are reported as a labeled pair rather than a single leakage number. The nine development probes, six target probes and three leakage probes on which any target-API emission is an error, add only suggestive elicitation evidence because their substring scorer is generous, their prompts share a scenario universe with the generator, and they are

too few to distinguish nearby arms. Retention is a separate construct measured on control benchmarks, not an implication of low API-benchmark churn.

Internal validity and causal attribution. Experiment versions were redesigned after diagnoses on the same benchmark, and checkpoint selection used its target slice. Pre-specified criteria, frozen final configurations, failed registered endpoints, and fresh-seed replication limit post hoc flexibility without making the final target set independent. Because package-level contrasts change several variables together, the SFT-versus-composition retention contrast does not identify effects of surface, subtraction, edit mass, rank, objective, or any individual recipe component.

Instrument validity. The benchmark asks only 224 of the 291 registered rows at L2, the 67 unasked rows all belonging to six APIs, so every twenty-five-API L2 rate uses the asked surface and the six APIs' evidence stays at L0. Official benchmark values check protocol plausibility without being point-estimate reproductions, IFEval is bit-exact under the serial client while a concurrent-client configuration moved identical-weight scores by up to 2.22 percentage points, and DS-1000 has aggregate outputs only and stays out of paired inference. Exact McNemar tests and Holm correction apply within the declared 24-, 24-, six-, and 12-test families.

Synthetic-data validity. The direct corpus audit establishes syntax, expected-API occurrence, live source-contract validation for the five scale-up packs, and complete prompt-answer separation from the benchmark, each at the corpus's achieved tier. The checks stop short of semantics: the lexical check does not resolve imports, receivers, or runtime bindings, runtime validation is not corpus-wide, and some historical corpora contain repeated templates. The manual reading described in Chapter 4, in which we read the generated examples individually and each corpus as a whole for validity, quality, and diversity, is a subjective judgment that cannot be reproduced or certified mechanically. Corpus quality and diversity claims beyond the mechanical tiers rest on that judgment. RQ2 therefore supports static, mechanically checked adequacy for the demonstrated adaptations, without a universal runtime-validity certificate.

External validity. All completed experiments use `Qwen2.5-7B-Instruct` and PyTorch, with no cross-model or cross-library replication, and the twenty-five-API extension stays on the same benchmark instrument. The target selection took 80% of the eligible new candidates (20/25) and retained five alternates, so the available pool does not support another increment of the same size under the same rules. Full release coverage, runtime routing, and deployment behavior remain untested, so the work provides validated building blocks instead of a library-version specialist.

8

Conclusion and Future Work

8.1. Conclusion

This thesis asked whether controlled LoRA fine-tuning on source-grounded synthetic data can reduce API hallucinations for post-cutoff APIs under fixed dependency constraints. A small LoRA adapter does make a missing API appear, the harder half of the problem is controlling where it appears, and the methodology built for that control carries from one API to five and then twenty-five on the LibEvoBench instrument [25], so the answer is conditional: adaptation is reliable only as far as conditional use, locality, and retention are measured and hold, and none of these follows from target acquisition or an aggregate score alone.

8.2. Answers to the Research Questions

1. **RQ1.** Benchmark-wide the errors split mainly between two nearly equal classes, wrong-but-valid choices of a semantic sibling or a deprecated ancestor of the gold API (27.58% of scored rows, `torch.chunk` completed as `torch.split`, `torch.linalg.svd` as the deprecated `torch.svd`) and phantoms that give real functionality a fabricated path, usually a truncation (27.24%, `torch.DataLoader` for `torch.utils.data.DataLoader`). Baseline hallucination is therefore the substitution of close names and the fabrication of plausible ones, and almost never a version error (0.36% anachronistic). On the post-cutoff targets the absence is complete on every target frame (0/12 broad and 0/30 allowlist for the controlled API, 0/141 at five APIs, and 0/224 asked rows at twenty-five), and the substitutes keep the required expression type while missing the symbol. The controlled target is completed by semantically nearby runtime checks, most often `torch.cuda.is_available` or `torch.jit.is_scripting`, and by two phantoms that put the target's own name under the wrong module (`torch.is_compiling`, `torch.dynamo.is_compiling`). The base model therefore lacks the API identity, not the task pattern, so adaptation must teach the missing identity and be judged by the outcome taxonomy together with the leakage pair, which catch a treatment that only trades one nearby wrong name for another. The answer covers API identity only, since signature validity and task-by-context variation were not evaluated, and it establishes behavioral absence in the operational sense of Chapter 1.

2. **RQ2.** Repairing that missing identity requires data that encodes it, and the corpora show that high-quality synthetic training data can be created from a teacher LLM working from each API's signature, documentation, and tests, since they pass complete syntax and expected-API checks and every configured live source contract, share no complete prompt-answer pair with the benchmark, and carry our recorded manual review of example quality and dataset diversity. The workflow is not yet a scalable option, because reaching that quality required iterative revision of the teacher prompt and manual per-example and per-dataset inspection. Since 62.58% of the audited rows are exact repeats from planned epoch expansion, and runtime and mechanical semantic-diversity evidence stays incomplete, the workflow certifies the mechanical tiers and separation while stopping short of a universal correctness certificate.

3. **RQ3.** With those corpora as input, controlled LoRA training can make one post-cutoff API producible, positive-only training leaks it across non-target rows, and anchors and hard negatives improve several matched frontiers without being universally required (the no-anchor diverse-format arm), so checkpoint selection must price leakage and non-target regressions alongside acquisition.

4. **RQ4.** Training task, data composition, and control design decide how much measured movement comes from target use, task calibration, and interference, the attribution that RQ4 isolates: knowledge-free controls reproduce broad gains with zero target acquisition, task-vector conclusions stay relative to their matched pair, the held-out probes stay development-only, and no single-API retention cell survives correction in its 24-test family.

5. **RQ5.** Under those controls, both method families scale beyond one API, with different replicated profiles, and on target rows each replaces phantom-heavy baseline errors with correct calls, the composition cutting the phantom share at twenty-five APIs to roughly a third of its baseline value. At five APIs the composition reaches a 75.2% three-seed mean against direct SFT's 28.4% peak mean (41.8% single-seed envelope), and at twenty-five, on the 224 rows the benchmark asks at L2, the composition reaches a replicated 62.8% and step-matched SFT 50.1%, with the new-API outcomes nearly seed-invariant at the composition champion. The one significant retention cost of the programme, the SFT package's GSM8K decline, appears at neither composition point, a package-level contrast whose cause the cross-method design cannot identify.

8.3. Future Work

Near term. The first line of work is the reinforcement-learning (RL) study that Chapter 7 specifies, GRPO- or DAPO-style group optimization whose rewards combine exact target identity, version validity, and format validity with penalties for resolved leakage, phantom paths, and regressions on baseline-correct rows, judged against a reward-ablated control under the same paired acquisition, leakage, churn, and retention gates. The completed single- and five-API surfaces already hold the baselines, adapters, and comparison points that study needs, so it runs there first. The second line replaces single teacher API calls with LLM agents that investigate a target API before writing examples for it, calling it under the pinned library version, executing the examples they write, and reading its implementation source in the library, so that high-quality training data is produced autonomously instead of through the teacher-prompt revision and manual inspection this thesis needed. Viability is testable now, by comparing agent-generated corpora against the manually inspected ones under the same audit tiers and the same downstream acquisition, leakage, and retention gates.

Medium term. If the near-term comparison favors it, locality-aware RL extends across many APIs, with the same gates applied per API. Scaling agent generation to larger API counts then requires automated quality and diversity checks that stand in for manual reading.

Long term. A library-version specialist requires release-scale coverage, runtime selection of the dependency version, validation of generated API behavior before code is accepted, rollback criteria, and monitoring after library updates. Cross-language evaluation and agentic repair workflows extend the same problem to larger software systems.

8.4. Takeaways for Practice

The completed evidence supports a short checklist for anyone fine-tuning version-specific API behavior. Do not train on positives alone, because without anchor examples that define where the new API is wrong, small adapters convert new symbols into global priors, and record which validation checks the training data actually passed. Do not select checkpoints on aggregate scores either: require per-API acquisition, the labeled leakage pair on non-target inputs, the failure-taxonomy shift, and paired regression and churn counts against the unmodified model. Evaluate retention as declared paired families within one harness whose base-model scores have been checked against published values, keeping aggregate-only tasks descriptive, and replicate the selected training configuration and operating point, because a single peak can misrepresent the distribution of exact-match conversion. At deployment,

verify emitted APIs against the installed version and monitor newly taught targets in unsupported contexts, since an adapter is not a runtime safety boundary.

Bibliography

[1] A. Vaswani et al., "Attention Is All You Need," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.

[2] W. X. Zhao et al., "A Survey of Large Language Models," *Frontiers of Computer Science*, vol. 20, p. 2 012 627, 2026.

[3] T. B. Brown et al., "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901.

[4] R. Bommasani et al., *On the Opportunities and Risks of Foundation Models*, arXiv:2108.07258 [cs], Aug. 2021.

[5] M. Chen et al., *Evaluating Large Language Models Trained on Code*, arXiv:2107.03374 [cs], Jul. 2021.

[6] A. Fan et al., "Large Language Models for Software Engineering: Survey and Open Problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, Melbourne, Australia: IEEE, May 2023, pp. 31–53.

[7] A. T. Kalai, O. Nachum, S. S. Vempala, and E. Zhang, *Why Language Models Hallucinate*, arXiv:2509.04664 [cs], Sep. 2025.

[8] B. Kanber, *All We Also Need Is ABSTAIN: Eliminating Hallucinations via a Single Token*, en, doi: 10.20944/preprints202510.1827.v1, Oct. 2025.

[9] L. Huang et al., "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, Mar. 2025.

[10] G. Hong et al., *The Hallucinations Leaderboard – An Open Effort to Measure Hallucinations in Large Language Models*, arXiv:2404.05904 [cs], Apr. 2024.

[11] Y. Bang et al., "HalluLens: LLM Hallucination Benchmark," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vienna, Austria: Association for Computational Linguistics, 2025, pp. 24 128–24 156.

[12] F. Liu et al., "Beyond functional correctness: Exploring hallucinations in LLM-generated code," *IEEE Transactions on Software Engineering*, vol. 52, no. 3, pp. 1037–1055, 2026.

[13] V. Agarwal, Y. Pei, S. Alamir, and X. Liu, *CodeMirage: Hallucinations in Code Generated by Large Language Models*, arXiv:2408.08333 [cs], Jul. 2025.

[14] Y. Tian et al., "CodeHalu: Investigating Code Hallucinations in LLMs via Execution-based Verification," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 24, pp. 25 300–25 308, 2025.

[15] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs," in *34th USENIX Security Symposium (USENIX Security 25)*, Seattle, WA: USENIX Association, Aug. 2025, pp. 3687–3706.

[16] A. Krishna, E. Galinkin, L. Derczynski, and J. Martin, *Importing Phantoms: Measuring LLM Package Hallucination Vulnerabilities*, arXiv:2501.19012 [cs] version: 1, Jan. 2025.

[17] Y. Wu, P. He, Z. Wang, S. Wang, Y. Tian, and T.-H. Chen, *A Comprehensive Framework for Evaluating API-oriented Code Generation in Large Language Models*, arXiv:2409.15228 [cs], Sep. 2024.

[18] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar, "On Mitigating Code LLM Hallucinations with API Documentation," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice*, 2025, pp. 237–248.

[19] Y. Chen, M. Chen, C. Gao, Z. Jiang, Z. Li, and Y. Ma, "Towards Mitigating API Hallucination in Code Generated by LLMs with Hierarchical Dependency Aware," in *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, Association for Computing Machinery, 2025.

[20] S. Kuhar et al., "LibEvolutionEval: A Benchmark and Study for Version-Specific Code Generation," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, L. Chiruzzo, A. Ritter, and L. Wang, Eds., Albuquerque, New Mexico, USA: Association for Computational Linguistics, Apr. 2025, pp. 6826–6840.

[21] T. Wu et al., *VersiCode: Towards Version-controllable Code Generation*, arXiv:2406.07411 [cs], Oct. 2024.

[22] N. Islah et al., *GitChameleon: Unmasking the Version-Switching Capabilities of Code Generation Models*, arXiv:2411.05830 [cs], Nov. 2024.

[23] D. Misra et al., "GitChameleon 2.0: Evaluating AI Code Generation Against Python Library Version Incompatibilities," in *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, M. Liakata, V. P. Moreira, J. Zhang, and D. Jurgens, Eds., San Diego, California, USA: Association for Computational Linguistics, Jul. 2026, pp. 46 792–46 831.

[24] C. Wang et al., "CodeSync: Synchronizing Large Language Models with Dynamic Code Evolution at Scale," in *Proceedings of the 42nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 267, PMLR, Jul. 2025, pp. 62 672–62 700.

[25] D. Cipollone, S. Titov, M. Izadi, E. Bogomolov, and A. van Deursen, *LibEvoBench: Probing Temporal Knowledge Stratification in Code Generation Models*, arXiv:2606.25402 [cs]. Accepted at the DL4Code workshop, ICML 2026, Jun. 2026.

[26] Qwen Team, *Qwen2.5 Technical Report*, arXiv:2412.15115 [cs], Dec. 2024.

[27] Qwen Team. "Qwen2.5-7B-Instruct model card," Accessed: Jul. 13, 2026. [Online]. Available: `https://huggingface.co/Qwen/Qwen2.5-7B-Instruct`.

[28] J. Li et al., "EvoCodeBench: An evolving code generation benchmark with domain-specific evaluations," en, in *Advances in Neural Information Processing Systems 37*, 2024.

[29] L. Zhang, W. Chen, L. Zhong, L. Peng, Z. Wang, and J. Shang, *Memorize or generalize? Evaluating LLM code generation with code rewriting*, arXiv:2503.02296 [cs] version: 1, Mar. 2025.

[30] E. J. Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," in *The Tenth International Conference on Learning Representations*, 2022.

[31] Anthropic. "Claude Sonnet 4.5," Accessed: Jul. 13, 2026. [Online]. Available: `https://www.anthropic.com/news/claude-sonnet-4-5`.

[32] Z. Gekhman et al., "Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?" In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds., Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 7765–7784.

[33] A. Goel, D. Schwartz, and Y. Qi, "Zero-knowledge LLM hallucination detection and mitigation through fine-grained cross-model consistency," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, Suzhou, China: Association for Computational Linguistics, Nov. 2025, pp. 1982–1999.

[34] T. Liu et al., "A Token-level Reference-free Hallucination Detection Benchmark for Free-form Text Generation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds., Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 6723–6737.

[35] A. Afzal, J. Vladika, and F. Matthes, *FActBench: A Benchmark for Fine-grained Automatic Evaluation of LLM-Generated Text in the Medical Domain*, arXiv:2509.02198 [cs] version: 1, Sep. 2025.

[36] H. Wu, Y. Yao, W. Yu, and N. Zhang, "ReCode: Updating Code API Knowledge with Reinforcement Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 40, no. 40, pp. 33 908–33 916, 2026.

[37] Z. Shao et al., *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*, arXiv:2402.03300 [cs], Apr. 2024.

[38] Q. Yu et al., "DAPO: An Open-Source LLM Reinforcement Learning System at Scale," in *Advances in Neural Information Processing Systems 38*, 2025.

[39] M. Sabbaghi, G. Pappas, A. Javanmard, and H. Hassani, *Robust Policy Optimization to Prevent Catastrophic Forgetting*, arXiv:2602.08813 [cs], Feb. 2026.

[40] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," in *Advances in Neural Information Processing Systems 36*, 2023.

[41] Q. Zhang et al., "Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning," in *The Eleventh International Conference on Learning Representations*, 2023.

[42] Y. Xiong and X. Xie, "OPLoRA: Orthogonal Projection LoRA Prevents Catastrophic Forgetting during Parameter-Efficient Fine-Tuning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 40, no. 40, pp. 34 088–34 096, 2026.

[43] D. J. Kopiczko, T. Blankevoort, and Y. M. Asano, "VeRA: Vector-based Random Matrix Adaptation," in *The Twelfth International Conference on Learning Representations*, 2024.

[44] Y. Sheng et al., "SLoRA: Scalable Serving of Thousands of LoRA Adapters," in *Proceedings of Machine Learning and Systems*, vol. 6, 2024.

[45] T. Mazzucotelli. "Griffe: Signatures for entire Python programs," Accessed: Jul. 14, 2026. [Online]. Available: `https://github.com/mkdocstrings/griffe`.

[46] Read the Docs, Inc. and contributors. "Sphinx-autoapi: A new approach to API documentation in Sphinx," Accessed: Jul. 14, 2026. [Online]. Available: `https://github.com/readthedocs/sphinx-autoapi`.

[47] G. Brandl and the Sphinx team. "Sphinx: Python documentation generator," Accessed: Jul. 14, 2026. [Online]. Available: `https://www.sphinx-doc.org`.

[48] G. Ilharco et al., "Editing Models with Task Arithmetic," in *The Eleventh International Conference on Learning Representations*, 2023.

[49] P. Yadav, D. Tam, L. Choshen, C. Raffel, and M. Bansal, "TIES-Merging: Resolving Interference When Merging Models," in *Advances in Neural Information Processing Systems 36*, 2023.

[50] L. Yu, B. Yu, H. Yu, F. Huang, and Y. Li, "Language Models are Super Mario: Absorbing Abilities from Homologous Models as a Free Lunch," in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 235, PMLR, 2024, pp. 57 755–57 775.

[51] M. S. Matena and C. A. Raffel, "Merging Models with Fisher-Weighted Averaging," in *Advances in Neural Information Processing Systems 35*, 2022.

[52] M. Bavarian et al., *Efficient Training of Language Models to Fill in the Middle*, arXiv:2207.14255 [cs], Jul. 2022.

[53] E. B. Wilson, "Probable Inference, the Law of Succession, and Statistical Inference," *Journal of the American Statistical Association*, vol. 22, no. 158, pp. 209–212, 1927.

[54] L. D. Brown, T. T. Cai, and A. DasGupta, "Interval Estimation for a Binomial Proportion," *Statistical Science*, vol. 16, no. 2, pp. 101–133, 2001.

[55] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.

[56] S. Holm, "A Simple Sequentially Rejective Multiple Test Procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[57] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Association for Computing Machinery, 2019, pp. 2623–2631.

[58] Y. Zheng, R. Zhang, J. Zhang, Y. Ye, and Z. Luo, "LlamaFactory: Unified efficient fine-tuning of 100+ language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 400–410.

[59] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems 36*, 2023.

[60] D. Hendrycks et al., "Measuring massive multitask language understanding," in *International Conference on Learning Representations*, 2021.

[61] Y. Wang et al., "MMLU-Pro: A more robust and challenging multi-task language understanding benchmark," in *Advances in Neural Information Processing Systems 37*, 2024.

[62] K. Cobbe et al., *Training verifiers to solve math word problems*, 2021. arXiv: 2110.14168.

[63] J. Zhou et al., *Instruction-following evaluation for large language models*, 2023. arXiv: 2311.07911.

[64] M. Chen et al., *Evaluating large language models trained on code*, 2021. arXiv: 2107.03374.

[65] Y. Lai et al., "DS-1000: A natural and reliable benchmark for data science code generation," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 202, PMLR, 2023, pp. 18 319–18 345.

[66] F. Petroni et al., "Language Models as Knowledge Bases?" In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 2463–2473.

[67] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How Can We Know What Language Models Know?" *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 423–438, 2020.

A

Supplementary Figures

This appendix collects supplementary views of the completed single-API, five-API, and twenty-five-API evidence reported in Chapter 6, together with the visual summary of the public-API extraction comparison of Chapter 4. The single-API taxonomy figure uses the matched broad artifacts named in Chapter 6, the reproducibility table reports the rerun pairs behind the retention-suite noise floors, and the five-API and twenty-five-API replication details and the scale-up taxonomy figure use their versioned thesis packs. The three L0 figures of Section A.6 are produced by the research repository's versioned analysis script from row-level scored outputs and imported unchanged, while the final extraction figure uses the frozen comparison described in Chapter 4.

A.1. Outcome Taxonomy Across Runs

Figure A.1 extends the curated taxonomy view of Chapter 6 to all main runs. The collapsed positive-only checkpoints are visible as a large anachronistic share, while the anchored adapters mainly convert phantom and no-API mass into correct and wrong-but-valid outputs.

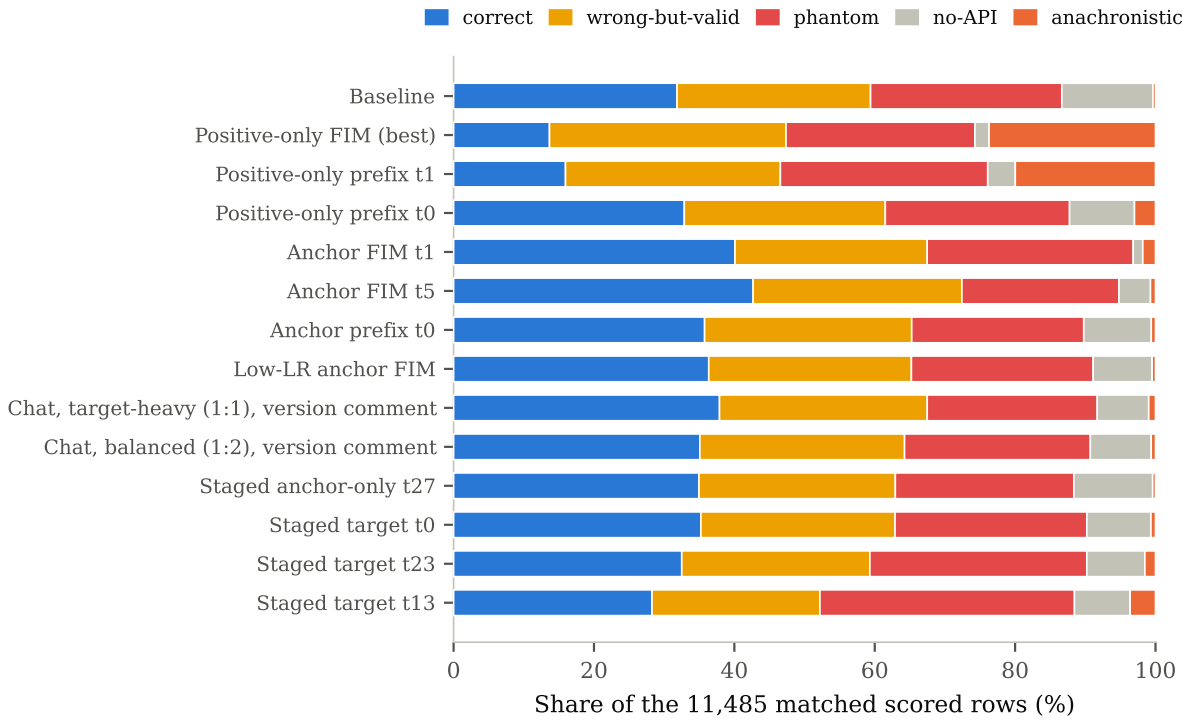


Figure A.1: Outcome taxonomy on the matched 11,485-row sample for the main runs. Categories follow the failure taxonomy of Chapter 4.

A.2. Reproducibility of the Control-Benchmark Suite

Table A.1 reports the rerun experiment behind the noise floors quoted in Chapter 5 for the retention suite. Under greedy decoding with a serial evaluation client, every same-weights rerun pair is exactly reproducible: not a single per-item outcome flips across runs executed on different days and devices, and two independent DS-1000 runs produce byte-identical generations for all 1,000 problems. Under a concurrent evaluation client, identical weights no longer reproduce exactly. On IFEval, the measured pairs flip 32 to 36 of 541 items each, with aggregate shifts between −1.85 and +2.22 percentage points and a mean near zero. For this reason, Chapter 6 uses the reproducible serial-client outcomes for IFEval arm deltas and does not interpret aggregate differences below the concurrent-client churn on the remaining generative lanes.

Table A.1: Same-weights, same-protocol rerun pairs of the retention suite. Wins and losses count items whose outcome flips between two runs of the identical model. A serial evaluation client reproduces exactly, while a concurrent client introduces symmetric churn.

Rerun pair (identical weights)	Client	n	Flips (w/l)	Δ (pp)
IFEval, base model, three run pairs	serial	541	0/0	0.00
IFEval, difference vector	serial	541	0/0	0.00
GSM8K (superseded protocol), two models	serial	1,319	0/0	0.00
HumanEval+ (superseded protocol), two models	serial	164	0/0	0.00
DS-1000, base model, generation bytes	serial	1,000	identical	0.00
IFEval, chat target-heavy arm	serial vs. concurrent	541	22/10	+2.22
IFEval, staged arm	serial vs. concurrent	541	17/17	0.00
IFEval, multitask arm	serial vs. concurrent	541	13/23	−1.85
IFEval, difference vector	serial vs. concurrent	541	18/14	+0.74

A.3. Replication Detail for the Five-API Operating Points

Table A.2 reports the per-seed values behind the replicated task-arithmetic champion of Chapter 6, including the control-free raw vector as a reference and the lower-λ knee composition carried by the retention suite, and Table A.3 gives the same view for the chat SFT lane's seed-specific peak checkpoints, where acquisition varies from 25 to 59 rows while the leakage pair stays within eight mention and ten resolved rows.

Table A.2: Per-seed values of the five-API task-arithmetic operating points. Exact target acquisition uses 141 rows. Leak gives the mention / resolved pair on the 11,436 off-target rows, and bc counts baseline-correct rows broken by the union of the pair. The knee (λ=0.9) is the retention family's second composition point.

Operating point	Seed	Acquisition	Leak (m / r)	bc
Raw, c=0, λ=1.0	42	72.3% (102)	201 / 157	29
	43	69.5% (98)	206 / 159	29
	44	73.8% (104)	214 / 165	29
Knee, c=0.25, λ=0.9	42	65.2% (92)	105 / 71	16
	43	65.2% (92)	104 / 75	15
	44	66.7% (94)	120 / 83	18
Champion, c=0.25, λ=1.0	42	76.6% (108)	171 / 123	26
	43	73.8% (104)	164 / 120	25
	44	75.2% (106)	175 / 128	27

Table A.3: Per-seed peak checkpoints of the chat SFT record recipe (cosine schedule) on the five-API surface. Acquisition uses the 141-row target surface, Leak gives the mention / resolved pair on the 11,436 off-target rows, and bc counts baseline-correct rows broken by the union of the pair.

Seed	Peak step	Acquisition	Leak (m / r)	bc
42	360	41.8% (59)	157 / 121	29
43	180	25.5% (36)	162 / 120	28
44	630	17.7% (25)	165 / 111	27

A.4. Replication Detail at Twenty-Five APIs

Table A.4 gives the per-seed values of the twenty-five-API task-arithmetic points, mirroring Table A.2. Figure A.2 shows the fourteen new APIs with L2 queries and lists the six structurally unqueried APIs separately. The SFT lane's step-matched replication values (111/114/112 at the champion step) appear in Chapter 6.

Table A.4: Per-seed values of the twenty-five-API task-arithmetic operating points on the 224-row asked surface, split into the new (83-row) and legacy (141-row) cohorts. Leak gives the mention / resolved pair on the 11,382 off-target rows, and bc counts baseline-correct rows broken by the union of the pair. The rank-32 raw vector (c=0) is the control-free reference, and the knee is the retention family's second composition point.

Operating point	Seed	Acquisition	New	Legacy	Leak (m / r)	bc
Raw, r=32, c=0, λ=1.0	42	67.0% (150)	44	106	407 / 396	96
	43	62.5% (140)	40	100	335 / 310	84
	44	66.1% (148)	43	105	370 / 354	86
Knee, r=16, c=0.25, λ=0.9	42	62.1% (139)	35	104	154 / 140	37
	43	62.1% (139)	35	104	148 / 135	32
	44	60.3% (135)	35	100	154 / 140	33
Champion, r=32, c=0.25, λ=1.0	42	65.2% (146)	40	106	355 / 342	81
	43	59.4% (133)	37	96	298 / 267	84
	44	63.8% (143)	42	101	350 / 333	84

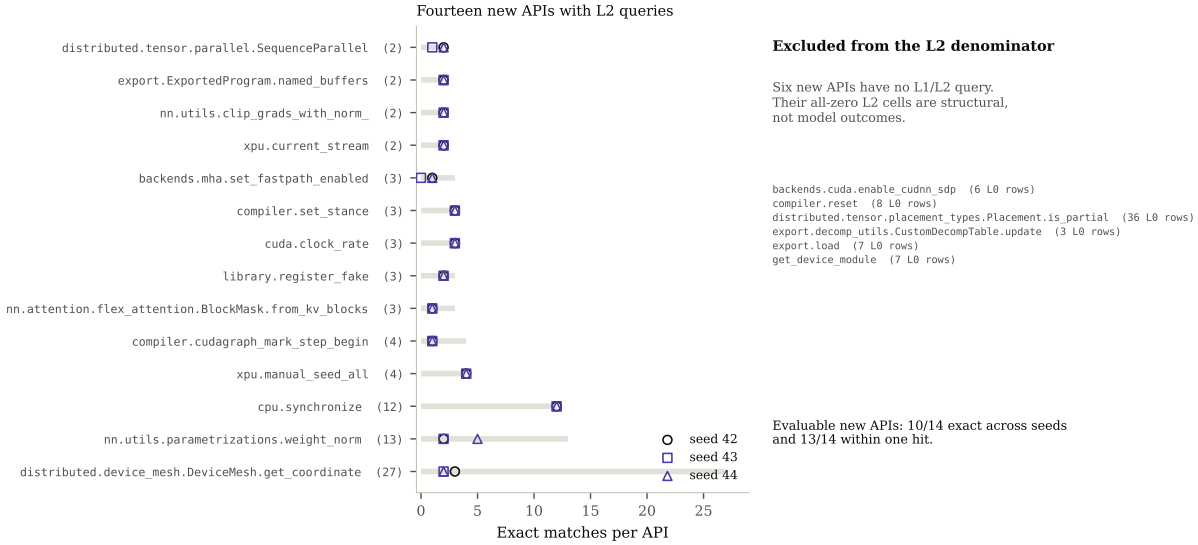


Figure A.2: Per-API exact matches of the twenty-five-API task-arithmetic champion across three seeds for the 14 new APIs with L2 queries. Grey bands span the queried rows, and overlapping markers indicate invariant counts. Ten APIs are exactly invariant and 13 remain within one hit. The six new APIs with no L2 query are listed separately and excluded from both denominators.

A.5. Outcome Taxonomy on the Scale-Up Target Surfaces

Figure A.3 gives the per-seed complement of the taxonomy figure of Chapter 6 (Figure 6.10): the full outcome-class decomposition of the scale-up operating points, in which the raw task vectors are control-free diagnostic references and the knee compositions are the retention family's second composition points. The twenty-five-API baseline bar is the task-arithmetic lane's served baseline, and the SFT lane's independently served baseline agrees with it within one row per class.

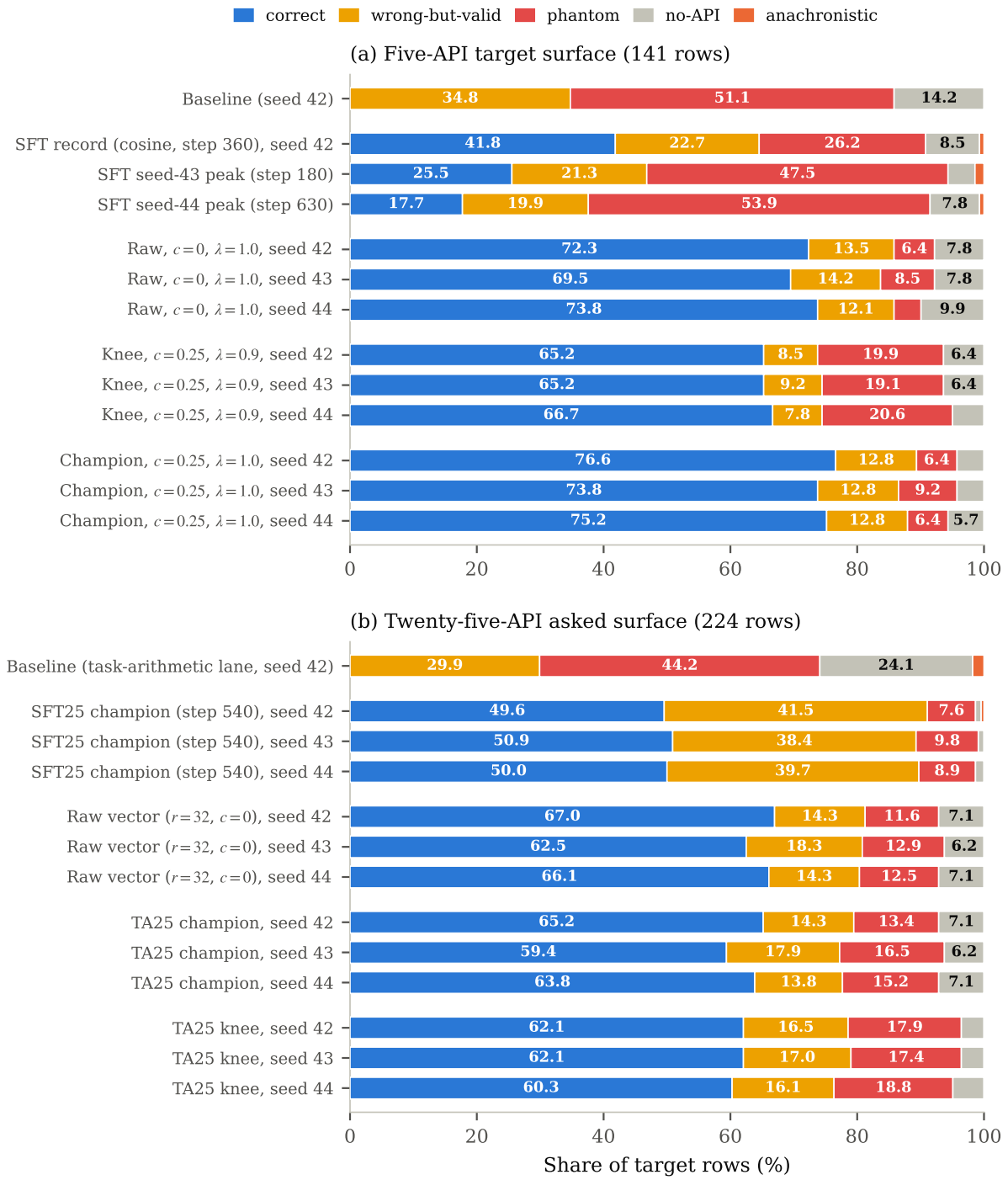


Figure A.3: Per-seed outcome taxonomy on the scale-up target surfaces, with each bar decomposed into the percentage shares of correct, wrong-but-valid, phantom, no-API, and anachronistic outputs. (a) Thirteen bars on the 141-row five-API target surface: the baseline, the SFT record (step 360, seed 42), the SFT seed-43 peak (step 180), the SFT seed-44 peak (step 630), then the raw (c=0, λ=1.0), knee (c=0.25, λ=0.9), and champion (c=0.25, λ=1.0) task-arithmetic points, each under seeds 42 to 44. (b) Thirteen bars on the 224-row asked surface of the twenty-five-API set: the baseline served by the task-arithmetic lane, then the SFT25 champion (step 540), the raw vector (r=32, c=0), the TA25 champion, and the TA25 knee, each under seeds 42 to 44.

A.6. Results at the Code-Only Context Level (L0)

Chapter 6 reports acquisition on the benchmark's scored context level L2, and its main text uses only an L0 summary for the six twenty-five-API targets without documentation at L2, while this section reports the full L0 evidence. L0 prompts contain only the code context, with no version requirement and no documentation, and L0 is the only level at which every target row (141 and 291) and every broad row is queried, which makes comparison possible for the six APIs skipped at L2. The champions remain those selected under the pre-registered L2 rule. Two independent baseline runs flip 138 of the 16,444 L0 broad rows, so smaller broad differences are not interpreted.

Figure A.4 gives the operating-point overview. The five-API task-arithmetic champion converts 76 of 141 rows at L0, compared with 108 at L2. The twenty-five-API task-arithmetic champion converts 118 of 291 at L0 across seed values 118/115/120, compared with 146 hits at L2 for the reporting seed. At twenty-five APIs, the SFT damage-priced champion reaches 148 of 291 at L0 across seed values 148/137/154 and exceeds the task-arithmetic point on this surface. This reversal is consistent with a difference in the rendered training prompts because the SFT packs include both documentation-bearing and code-only completion shapes, while the frozen task-arithmetic recipe renders only the documentation-bearing shape. The large `Placement.is_partial` slice also matters: SFT solves 36 of 36 rows and task arithmetic solves 2, while task arithmetic leads 116 to 112 after those rows are removed. These observations describe a surface association and do not isolate the training format as its cause.

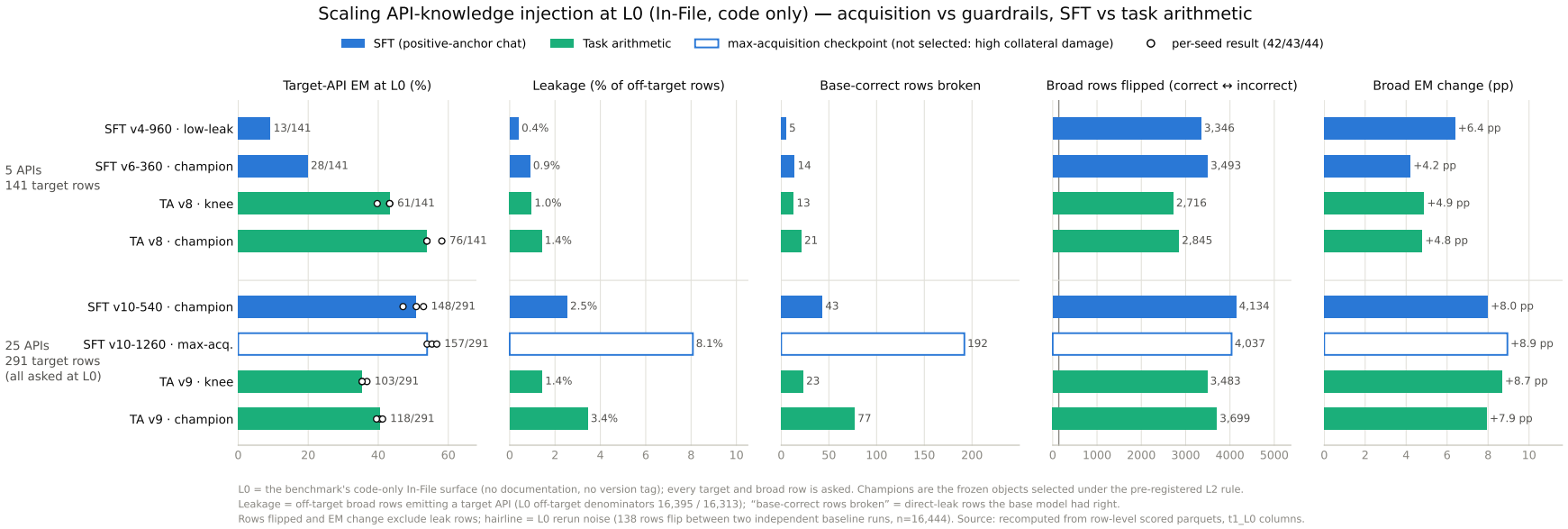


Figure A.4: Operating-point overview at L0, where every target and broad row receives a code-only prompt. Bars are the L2-selected champions re-measured at L0, and open markers give per-seed values where replication exists. Leakage, broken baseline-correct rows, flipped rows, and broad exact-match change use the L0 off-target surfaces of 16,395 rows at five APIs and 16,313 at twenty-five APIs. The hairline marks the measured L0 rerun floor of 138 flips among 16,444 rows.

Figure A.5 resolves the twenty-five-API L0 comparison per API. The six documentation-empty APIs (marked with a dagger) are not systematically harder at L0 than the asked nineteen: all six reach nonzero L0 acquisition under the SFT champion and five of the six under both methods, and the largest of them, `Placement.is_partial`, is fully acquired by the SFT champion. The per-API view also shows that the legacy receiver method `get_local_rank` inverts between the lanes at L0 (SFT 0 of 45, composition 8 of 45) while both lanes agree exactly on the small saturated APIs, the same binding-level agreement the L2 seed-invariance analysis found.

Figure A.6 completes the L0 view with the answer-class transition matrices of the four scale-up champions on the non-leak off-target broad rows, the L0 analogue of the taxonomy analysis of Chapter 6. The transition structure matches the L2 story: most movement exchanges wrong-but-valid, phantom, and no-API mass in both directions, with the net improvement concentrated in phantom-to-correct and no-API-to-correct flows, and no anachronistic buildup.

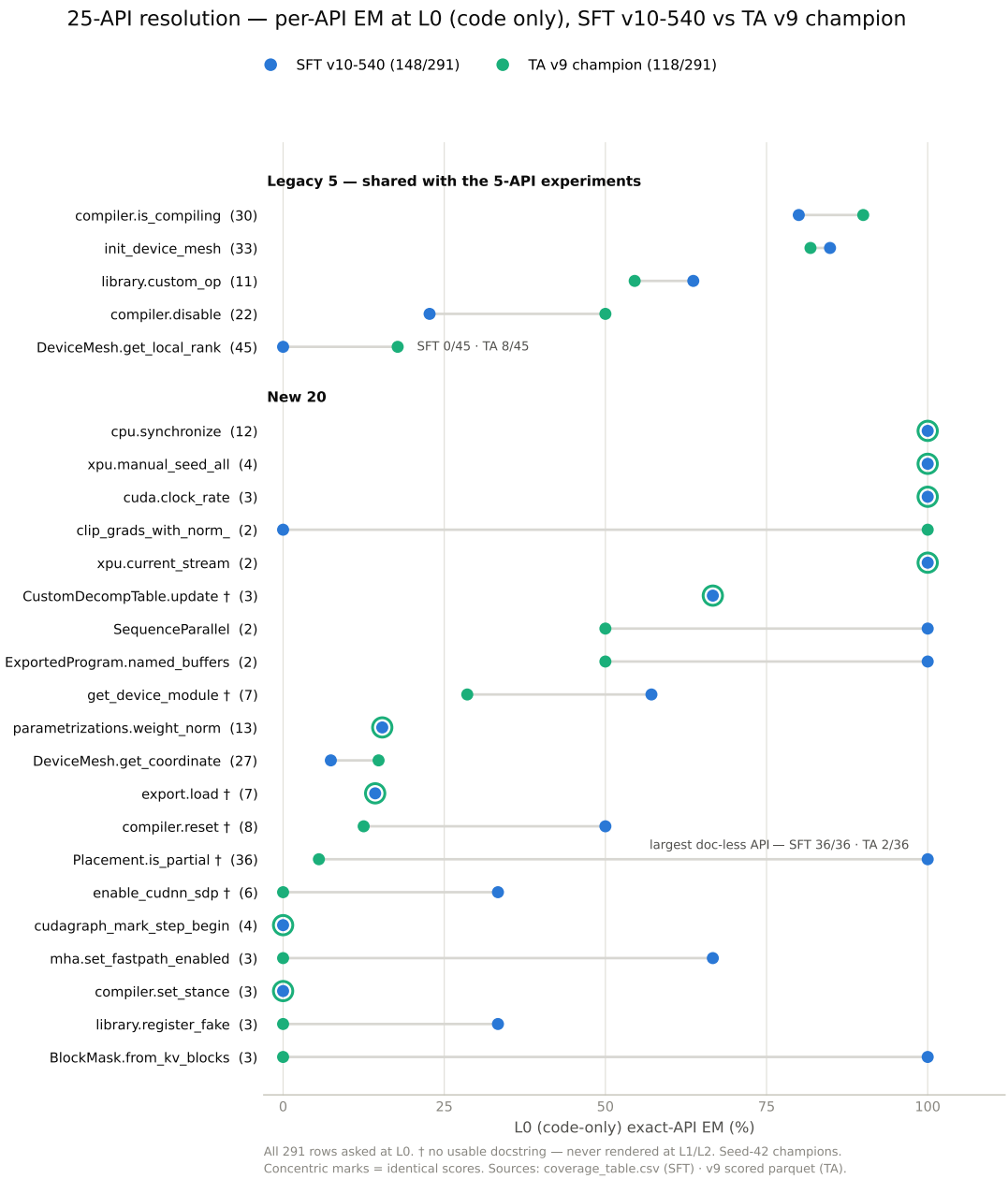


Figure A.5: Per-API exact match at L0 for the twenty-five-API champions of both lanes under seed 42. Daggers mark the six documentation-empty APIs that the benchmark never queries at L1/L2, and concentric markers indicate identical scores.

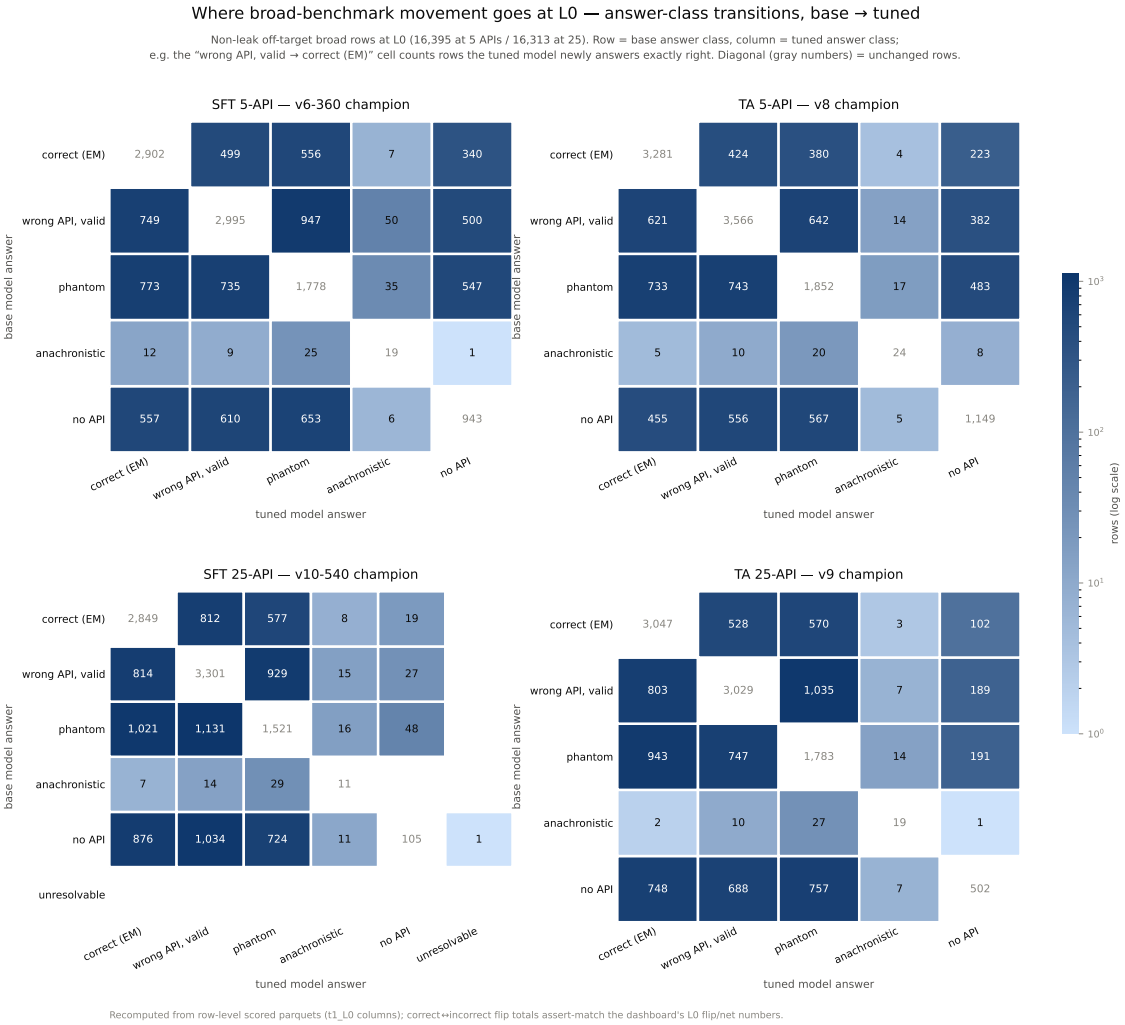


Figure A.6: Answer-class transitions from the base model to each scale-up champion on the non-leak off-target broad rows at L0. Counts use a logarithmic scale, and diagonal cells are unchanged rows.

A.7. Public-API Extraction Comparison

Figure A.7 visualizes the extraction comparison tables of Chapter 4: the extractor families differ by more than an order of magnitude in surface size, and the Sphinx-derived set behaves as a high-precision, low-recall subset of the runtime-visible surface while capturing a substantial share of the cross-method consensus core.

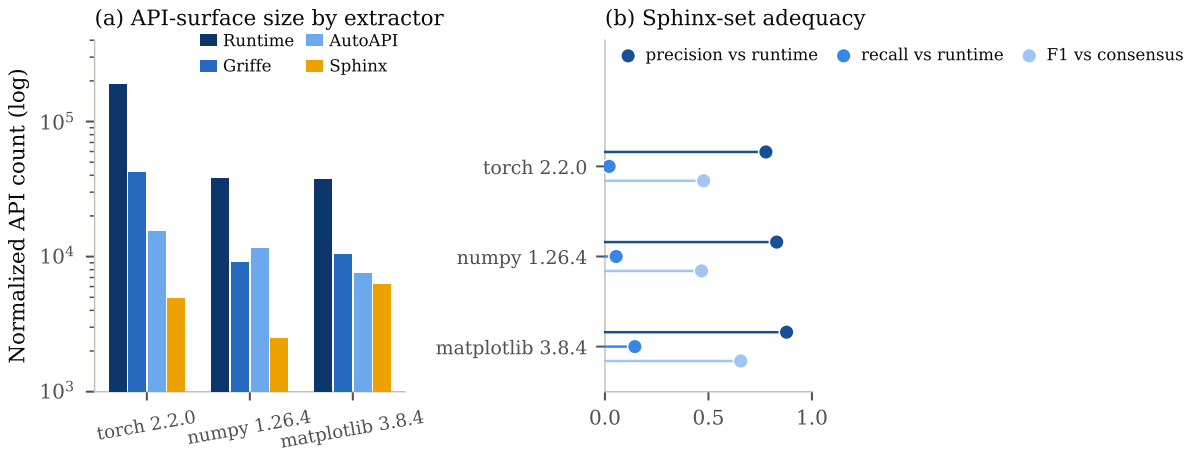


Figure A.7: Public-API extraction comparison on the three pinned library versions of Chapter 4. (a) Normalized API counts per extractor on a logarithmic scale. (b) Precision and recall of the Sphinx-derived set against runtime introspection, and its F1 against the multi-method consensus set.

B

Use of Generative AI Tools

Generative AI tools supported two kinds of work in this project, running the GPT-5.1, GPT-5.2, GPT-5.4 and GPT-5.6-Sol models on the OpenAI side and the Claude Opus 4.5 and Claude Fable 5 models on the Anthropic side. Conversational assistants, ChatGPT and Claude, answered comprehension questions about papers and technologies during the literature study and the design of the experiments, with prompts of the form "Explain concept X in this paper for me". Agentic coding tools, Codex and Claude Code, aided in the development of the code of the project under our direction, with prompts of the form "Implement a method that does X", "Implement a script that does X", and "Change the implementation of X such that Y". The same agentic tools produced analysis figures on request, with prompts of the form "Generate a plot with the data from experiment X", and retrieved literature metadata, with prompts of the form "Find me the most recent published version of paper X and give me a citation for it".

Agentic tools also assisted the writing of this document under our direction, with prompts of the form "Format this table/figure/paragraph such that it renders nicely in the page" and "Reformulate this paragraph such that it follows the following main ideas". All AI output, whether text, code, figures, or retrieved references, was verified by the author, who takes full responsibility for the content of this thesis.