Procedural Content Metageneration via Program Search and Continual Abstraction Discovery

Matthew Siper
Game Innovation Lab
New York University
New York, USA
ms12010@nyu.edu

Ahmed Khalifa
Institute of Digital Games
University of Malta
Msida, Malta
ahmed@akhalifa.com

Julian Togelius
Game Innovation Lab
New York University
New York, USA
julian@togelius.com

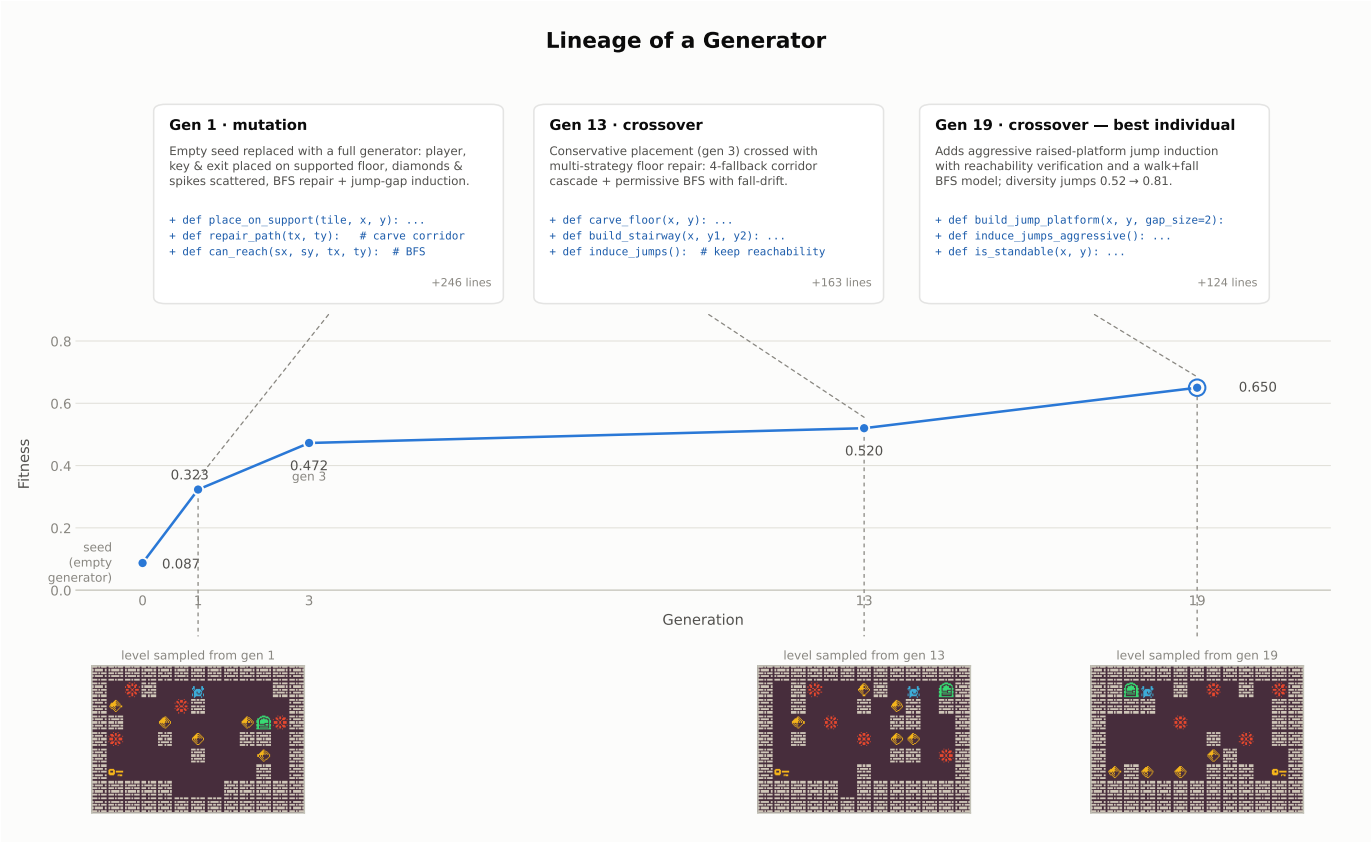

Fig. 1. Evolutionary lineage of the best Dangerous Dave generator. Each callout summarizes the code introduced at that step, and each image shows a level sampled from the corresponding generator. The new generators just need to be better than the starting seed to be accepted in the archive.

***Abstract*—Large language models can generate executable programs, which makes it possible to search directly over procedural content generators rather than individual levels. We study this approach in Sokoban, Zelda, Dangerous Dave, and Lode Runner. Each run evolves complete Python generators through language-model mutation and crossover. We introduce Continual Abstraction Discovery, or CAD, which extracts reusable primitives from high-fitness programs into a run-specific helper module. A 2×2 experiment crosses CAD with access to a fixed hand-written domain API. The completed data set contains 160 complete runs, with at least ten 50-generation runs in every cell. CAD raises mean final best fitness in all eight domain and API comparisons. Across all CAD runs, learned libraries are adopted by most later programs and repeatedly rediscover validation, reachability, and structural utilities. These results support that discovering reusable primitives improves evolutionary program search for content generators.**

***Index Terms*—Large Language Models, Procedural Content Generation, Level Generation, Program Synthesis, Code Generation, Genetic Programming, Evolutionary Search**

979-8-3315-9476-3/26/$31.00 ©2026 IEEE

I. Introduction

Games that feature procedurally generated content usually rely on custom generators tailored to the mechanics, constraints, and design goals of a particular game. Creating these generators requires substantial engineering and repeated evaluation. Procedural content generator generation seeks to automate this process by searching for the generator itself rather than directly searching for individual artifacts [1]. This process can also be called procedural content *metageneration*.

Existing approaches learn generators through evolutionary computation, machine learning, or reinforcement learning [1]–[5]. Many produce neural or otherwise opaque representations. Such generators can be effective, but they are difficult to inspect and edit. Executable source code offers a different representation. A program can be compiled, tested, modified by a designer, and reused outside the search system.

Large language models are increasingly capable of writing and revising code. This makes it possible to place a code-generating model inside an evolutionary loop. A candidate generator is executed, its levels are evaluated, and the resulting feedback guides later mutation and crossover calls. Related systems have used language models as variation operators for executable program search [6]–[8]. Procedural content metageneration presents a demanding instance of this problem since the quality of a generator can only be judged through repeated execution and behavioral evaluation.

Search over raw code fixes the vocabulary available to variation. As programs grow, useful routines for reachability, entity normalization, repair, and structural construction may be independently recreated in many candidates. A hand-written helper API can expose reusable operations, although its vocabulary remains fixed before search begins. We introduce *Continual Abstraction Discovery*, or CAD, as a way to let the set of executable primitives increase during a run. CAD extracts reusable utilities from high-fitness programs into a run-specific helper module, validates the module, and refactors programs to use the learned utilities when this can be done safely.

We evaluate the system on four domains from the PCG Benchmark [9]. The main study crosses a fixed domain helper API with CAD. This design separates access to a reusable vocabulary from the discovery of a vocabulary during search.

We make four contributions.

- We demonstrate LLM-driven evolutionary search over executable Python-level generators in Sokoban, Zelda, Dangerous Dave, and Lode Runner.
- We introduce CAD, a search-time mechanism that extracts, validates, and reuses helper functions discovered within high-fitness programs.
- We present a 2×2 study that separates CAD from access to a fixed expert API across 160 completed runs.
- We show positive CAD effects across the 4 game domains and provide an analysis of the learned-library growth, adoption, and primitives.

II. Related Work

A wide variety of computational methods have been applied to generating game content, in particular, game levels. These range from evolutionary computation [10] and constraint satisfaction [11] to supervised learning [3] and reinforcement learning [4]. Published games typically rely on constructive generators [12] specialized for the particular game, which limits reusability.

Recently, researchers have applied LLMs to generating game levels [13]–[20]. The results are mixed. In order to represent a game level in a way that an LLM can understand, it needs to be converted to a string somehow. Typically, this means representing the level as a sequence of rows, where each character represents a tile. Initial results showed that fine-tuning generators on levels from particular games improved generation capacity, and that more modern and recent LLMs outperform older and smaller ones on level generation (as on most other things) [13].

However, current LLMs have limitations when generating game content, particularly spatial content such as levels. For example, LLMs tend to have problems ensuring that clear paths exist from point A to point B, and that the right number of items and characters of various types exist (such as exactly one player character, and the same number of doors as keys). Other pathologies of LLM-generated game levels seem to originate in weak spatial reasoning [13], [16], [18].

One of the major use cases of LLMs is code generation. Software engineering is currently being transformed by the emergence of LLMs that can produce working code from an informal specification. However, in many cases, you know what you want but not exactly how to get there. In those cases, you can create a fitness function measuring what you want, and then embed the code-generating LLM inside an evolutionary loop. This approach was popularized by AlphaEvolve [6], which, among other things, discovered an improved matrix multiplication algorithm, and similar methods have also been applied to creating scientific data analysis solutions and financial trading policies [7], [8].

Using LLMs to generate code inside an evolutionary loop can be seen as a form of genetic programming, which is the application of evolutionary computation to program synthesis [21], [22]. From this perspective, the LLM is carrying out the mutation and/or crossover operations in genetic programming. An important concept in genetic programming is Automatically Defined Functions, where a set of callable functions is evolved in tandem with the main program, constituting a form of evolved abstraction [23].

Continual abstraction discovery (CAD) is also related to recent work on agent skill libraries, where reusable code or behaviors are stored for later use, such as Voyager's

skill library for Minecraft agents [24]. The key difference in our setting is that the library is not a record of solved tasks or manually exposed skills, but a continually updated set of helper functions extracted from high-performing generator programs during an ongoing evolutionary search. CAD, therefore, functions as a search-time representation mechanism. It changes the program primitives available to later mutation, crossover, and refactoring calls while the population is still being optimized.

Returning to PCG research, work on PCG through machine learning [3], including reinforcement learning [4], generally learns generators, which are then used to generate game content. However, they mostly learn neural networks or other types of black box representations, which do not allow for easy human interpretation and editing. There are exceptions to this, where evolution or similar methods are used to learn generators expressed in a symbolic representation [1], [2]. The approach presented in this paper can be seen as procedural procedural content generator generation, or procedural content metageneration, using a search-based approach in generator space enhanced by LLM code writing. A key invention beyond the combination of tool and domain is the Continual Abstraction Discovery mechanism, which is similar to Automatically Defined Functions but using LLM-driven evolution.

III. Method

A. Program Representation

Each individual is a complete Python program defining `generate(context_dict)`. The function receives the grid dimensions and an initial level and returns a tile grid. A program may contain local functions, constants, and layout routines, while the public interface remains fixed across the population. This representation allows every candidate to be compiled, executed, inspected, and evaluated with the same harness.

All runs begin from one minimal seed program. The seed contains tile constants and a blank or copied initial grid, with no generation logic. It is evaluated once and becomes the root of the archive. The archive retains every successfully evaluated individual.

B. Evolutionary Search

At each generation, one parent is sampled from the archive with fitness-proportional probability. Fitness values are floored at 10^{-6} before normalization. When the archive contains at least two individuals, crossover is selected with probability 0.25. A second parent is then sampled with the same rule. Failed crossover attempts fall back to mutation.

Mutation and crossover are performed by an LLM. The prompt contains the parent program or programs, a domain description, the accumulated run memory, and the fixed generator contract. Mutation asks for a strong behavioral edit. Crossover asks the model to combine compatible mechanisms from both parents into one complete executable program.

Every candidate passes through a robustness pipeline. The program must compile and complete at least three of five smoke-test executions. A failure triggers an LLM correction call conditioned on the error. The mutation or crossover cycle is attempted up to four times. A generation that never produces a valid candidate is recorded with fitness zero. Parse failures, compile failures, smoke-test failures, and timeouts are logged separately.

After every generation, an LLM reflection call appends a structured entry to a per-run memory file. Each entry records the attempted change, the observed result, and global lessons for later search. The accumulated global lessons are supplied to subsequent mutation and crossover calls in place of raw per-parent evaluation feedback.

C. Evaluation and Fitness

Each candidate is executed 30 times. The sampled levels are evaluated with the quality, validity, and diversity functions supplied by the PCG Benchmark [9]. We compute the fraction of fully valid levels, the mean benchmark quality, and the mean pairwise diversity over all sampled level pairs. Fitness is defined as

$$F = \frac{1}{3}\left(V + Q + D\right), \tag{1}$$

where V is the fraction of levels whose quality is equal to 1.0, Q is mean level quality, and D is mean pairwise diversity. All terms lie in $[0, 1]$. Candidates that crash or fail validation receive fitness zero.

D. Continual Abstraction Discovery

One of the contributions of this work is continual abstraction discovery. Starting at generation 8 and at five-generation intervals thereafter, the system identifies programs at or above the 75th percentile of fitness within the run. An LLM extraction call analyzes these programs and proposes reusable utility functions from that code. An extracted function must compile and pass a smoke test. The system allows up to two compile corrections. The source program is then refactored to call the new helpers. A refactor is accepted only when the refactored program matches the original program on a fixed-seed behavioral test (the generated levels has similar behavior characteristics). Accepted refactors are re-evaluated and their archive entries are updated. Refactoring is disabled when its success rate remains below 30% after a ten-generation warmup.The extracted functions are recorded in a system library that can be used by programs during evolution.

IV. Experimental Design

A. Domains

We evaluate four tile-based domains from the PCG Benchmark:

- **Sokoban:** is a 5x5 Japanese puzzle game where the player needs to push boxes to specific target locations.
- **Zelda:** is a 7x11 dungeon crawler inspired by dungeon rooms from the original The Legend of Zelda game.

The goal of the level is to get a key and reach the exit without getting killed by monsters.

- **Dangerous Dave:**is a 7x11 platformer game based on game with the same name. The goal of the game is for the player to get a key and reach the goal without getting killed by spikes.
- **Lode Runner:** is a 11x16 puzzle platformer game based on a game with the same name. The goal of the game is for the player to collect all the gold and avoid getting caught by the enemies. The puzzle element is that the player can't jump; they can only navigate the level by walking, digging through the floor, climbing ladders, and moving through ropes.

For all these problems, we use the framework's quality and diversity outputs. These outputs mainly use an A^* solver to compute quality and use some behavioral and visual characteristics to compute diversity. Each call to a generator is limited to two seconds.

B. Factorial Conditions

The main study tests 4 experiments that test the usefulness of having helper functions. To assess this, we have two parameters that we are testing:

- **CAD:** allowing the LLM to abstract programs and generate helper functions as explained in section III-D.
- **Expert API:** having a group of hand-designed helper functions that are commonly used in PCG. The library contains validated primitives for entity normalization, reachability, repair, and domain-specific structural operations. Examples include `connect_floors_with_ladder` and `ensure_one_player`. The expert API tests whether providing helper functions will be enough or CAD is still needed.

Every experiment uses the same minimal seed, evolutionary search, reflection memory, correction pipeline, evaluation budget, and fitness function. We ran each experiment 10 times for 50 generations.

C. LLM and Execution Configuration

All language-model calls use GLM 5.2. Mutations uses temperature 0.2. Correction and refactoring use temperature 0.1. The mean 50-generation run used approximately 9.5 million input tokens and 2.6 million output tokens. The mean cost was approximately $25 and the mean wall-clock time was approximately 2.5 hours. Each run stores per-generation statistics, program sources, evaluation analyses, prompts, valid rendered levels, the evolution tree, reflection memory, helper-module snapshots for CAD conditions, and token and timing logs.

V. Results

A. Cross-Domain Search Performance

Figure 2 shows a positive endpoint difference for CAD in every domain. The magnitude varies across games. The Base contrast is small in Lode Runner and larger in Sokoban. Under the fixed API, the largest separation occurs in Lode Runner, where the CAD trajectory continues improving while the no-CAD trajectory plateaus earlier. All experiments favor CAD, which gives an exact two-sided sign-test result of $p = 0.008$. An interesting outcome is that only in Zelda, having a predefined API is better than having no API to start with. We believe that the simplicity of the problem benefited from the available functions, as the other problems need more complex solvers than just pathfinding and counting objects.

Figure 3 shows example of the generate levels for the best generator for every game. Looking at the generated levels, we can notice that the visual diversity of levels, especially in Dangerous Dave, is lower with CAD than without. We believe the reason behind that is that the diversity metric of Dangerous Dave only cares about the trajectory of the player solution and not the visual diversity, which makes these levels diverse even if they look similar.

TABLE I
Most frequently rediscovered learned abstractions across all CAD runs. Semantically equivalent functions are grouped, and Total Calls is the aggregate call count in the supplied audit. The complete audit is provided in the supplementary source.

Abstraction	Category	Total Calls
In Bounds	Other	4,693
Entity Count Normalization	Validation	3,848
Ensure One Player	Validation	2,611
Grounded Empty Cells	Utility	465
Flood Fill	Other	569
Deep Copy Grid	Other	118
Find Player Position	Utility	2,370
Ladder Placement	Structural	768
Reachability BFS	Validation	344
Relocate Unreachable Entities	Validation	320
Connected Components	Other	297
Find Tiles	Other	574

B. Learned-Library Growth and Adoption

Figure 4 shows that the learned libraries expand sharply during the early extraction cycles and then stabilize across all four domains. Once helper use begins, the reported adoption rate remains above roughly 80% for most later generations, while programs contain approximately 15 to 20 helper calls on average.

The program-size comparison shows a second effect. In the Base condition, CAD reduces the mean length of the best program from roughly 370 lines to roughly 296 lines. Programs with the fixed API are already much shorter, at roughly 160 lines without CAD and 173 lines with CAD. This pattern is consistent with reusable logic moving out of the main generator program. The fixed API absorbs much of this complexity from the beginning, leaving less source-length reduction for CAD.

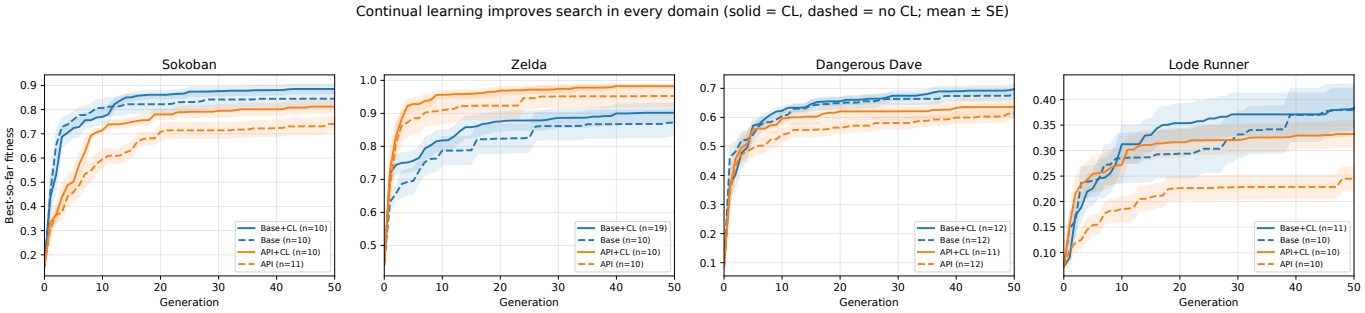

Fig. 2. Best-so-far fitness over 50 generations in each domain. Solid lines show CAD and dashed lines show no CAD. Shading reports mean ± standard error, and legends show the completed run count for each cell. The plot uses CL as the implementation label for CAD.

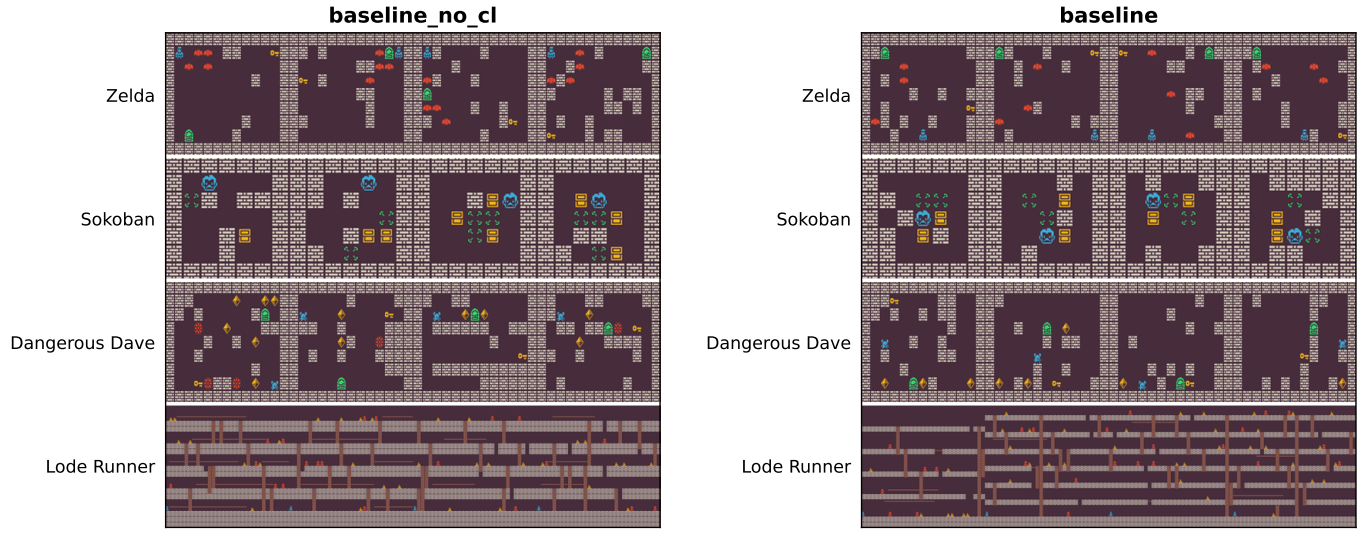

Fig. 3. Representative generated levels from the Base condition without CAD on the left and with CAD on the right. Rows show Zelda, Sokoban, Dangerous Dave, and Lode Runner. Each row contains four samples. The images illustrate output structure and are not used as evidence of playability, which is measured through benchmark execution.

C. Repeated Abstraction Discovery

Table I shows a recurring core of validation, reachability, and structural operations. `in_bounds` and Entity Count Normalization each appear in 28 of the CAD runs. Ensure One Player and Grounded Empty Cells appear in 24 runs. Find Player Position, Ladder Placement, Reachability BFS, Relocate Unreachable Entities, and `connected_components` each appear in 20 runs. CAD discovers both a shared set of broadly useful operations and a large tail of run-specific utilities. The repeated functions correspond to common generator requirements such as entity counts, traversal, connectivity, and tile placement.

Figure 5 shows the effect of CAD on the generated levels. For Lode Runner, it abstracted a function that put enemies beside gold to make the levels harder, where the next upcoming generators will make sure to place enemies beside the gold. For Dangerous Dave, something similar happen where CAD added a function that adds enemies/harmful objects around the solution path to make the level harder, which affects the upcoming generators that start adding these objects.

VI. Discussion

A. Cross-Domain Evidence

The results show that one evolutionary program-search framework can operate across four games with different grid sizes and benchmark evaluators. CAD improves the mean fitness in every domain. The largest effect appears in Lode Runner when the expert API is available. Search already has domain-specific callable routines in this condition, yet the run-specific learned vocabulary still improves the best fitness. The three other API cells also favor CAD, although their tests do not reach $p < 0.05$.

B. CAD as Representation Adaptation

CAD changes the operations available to later variation calls during search. The executable substrate remains Python, while the effective vocabulary grows as helpers are extracted. The mechanism measurements show that the resulting functions are adopted by later programs and called

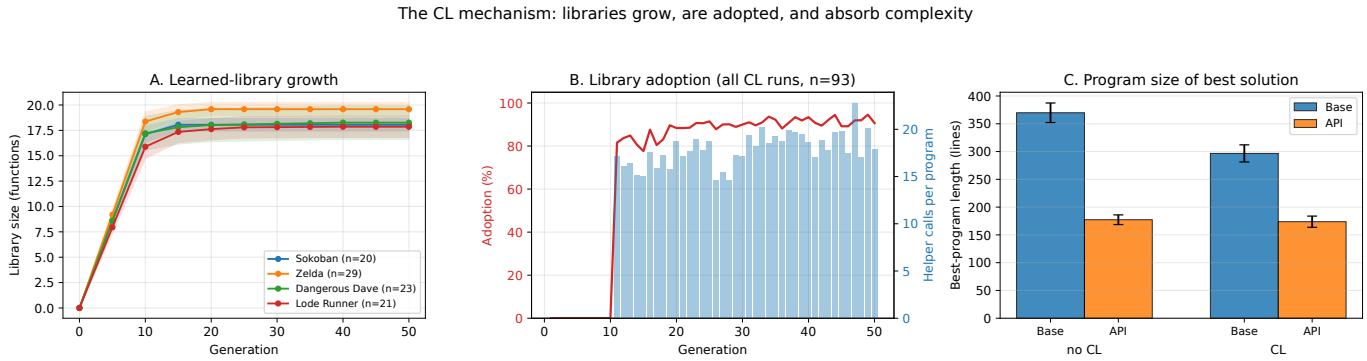

Fig. 4. Mechanism measurements across CAD runs. Panel A shows learned-library growth by domain. Panel B shows the reported adoption rate and helper calls per program. Panel C compares the line count of the best solution under Base and API prompting with and without CAD. Shading and error bars report uncertainty around the mean. The plot uses CL as the implementation label for CAD.

Fig. 5. Two primitives from the evolved helper library, each applied to a level (before, left; after, right; changed tile circled): guard_gold places an enemy beside a gold piece in Lode Runner, and place_enemies_away_from_critical places an enemy off the player → key → door critical path (white dots) in Zelda.

repeatedly. Commonly recurring helper functions address patterns such as entity normalization, player placement, reachability, connectivity, and structural construction. These are central operations in tile-based generator design. The large run-specific tail also indicates that CAD does not converge to one fixed library. It combines a recurring core with utilities shaped by the domain and search history.

The program-size analysis supports a related interpretation. CAD shortens the best Base programs, while the fixed API already yields compact source libraries. This does not establish that shorter programs are intrinsically better. It shows that reusable callable vocabularies can offload logic out of the main generator, which may make later edits easier to express and preserve.

C. Practical Implications and Limitations

Solvability, benchmark quality, and diversity do not fully capture visual style, pacing, novelty, or designer intent. Human review remains important when the desired

generator has aesthetic or experiential goals that are absent from the benchmark.

The compute cost is substantial. A typical 50-generation run uses millions of tokens of a near-frontier-class model and several hours of wall-clock time. CAD introduces additional extraction, correction, and refactoring calls. Future work should evaluate fitness gains against token use and wall-clock cost.

CAD is evaluated as a complete pipeline. Helper extraction, module correction, and source refactoring are not separated into individual ablations. The mechanism traces show growth, adoption, and program-size changes, although they do not isolate the causal contribution of each component. Additional work should test longer budgets, other language models, non-tile content, cross-run library transfer, and designer editing of discovered helpers.

VII. Conclusion

We presented an evolutionary program-search system that uses an LLM to generate executable procedural content generators. The evaluation covers Sokoban, Zelda, Dangerous Dave, and Lode Runner, where each one is run 10 times. Continual Abstraction Discovery raises the mean final best fitness regardless you start with an empty helper library or expert helper functions.

Across the CAD runs, learned libraries grow, are adopted by later programs, and repeatedly recover validation, reachability, and structural utilities. The results support CAD as a practical mechanism for adapting the searchable program vocabulary during procedural content metageneration.

References

[1] M. Kerssemakers, J. Tuxen, J. Togelius, and G. N. Yannakakis, "A procedural procedural level generator generator," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2012, pp. 335–341.

[2] A. Khalifa and J. Togelius, "Multi-objective level generator generation with marahel," in *Proceedings of the 15th International Conference on the Foundations of Digital Games*, 2020, pp. 1–8.

[3] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[4] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, 2020, pp. 95–101.

[5] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Computing and Applications*, vol. 33, no. 1, pp. 19–37, 2021.

[6] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. Mehrabian *et al.*, "Alphaevolve: A coding agent for scientific and algorithmic discovery," *arXiv preprint arXiv:2506.13131*, 2025.

[7] E. Aygün, A. Belyaeva, G. Comanici, M. Coram, H. Cui, J. Garrison, R. J. A. Kast, C. Y. McLean, P. Norgaard, Z. Shamsi *et al.*, "An ai system to help scientists write expert-level empirical software," *arXiv preprint arXiv:2509.06503*, 2025.

[8] M. Siper, A. Khalifa, L. Soros, M. Nasir, and J. Togelius, "Profit: Program search for financial trading," *Available at SSRN 5889762*, 2025.

[9] A. Khalifa, R. Gallotta, M. Barthet, A. Liapis, J. Togelius, and G. N. Yannakakis, "The procedural content generation benchmark: an open-source testbed for generative challenges in games," in *Proceedings of the 20th International Conference on the Foundations of Digital Games*, 2025, pp. 1–12.

[10] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[11] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187–200, 2011.

[12] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[13] G. Todd, S. Earle, M. U. Nasir, M. C. Green, and J. Togelius, "Level generation through large language models," in *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 2023, pp. 1–8.

[14] S. Sudhakaran, M. González-Duque, M. Freiberger, C. Glanois, E. Najarro, and S. Risi, "Mariogpt: Open-ended text2level generation through large language models," *Advances in Neural Information Processing Systems*, vol. 36, pp. 54 213–54 227, 2023.

[15] C. Hu, Y. Zhao, and J. Liu, "Game generation via large language models," in *2024 IEEE Conference on Games (CoG)*. IEEE, 2024, pp. 1–4.

[16] R. Gallotta, G. Todd, M. Zammit, S. Earle, A. Liapis, J. Togelius, and G. N. Yannakakis, "Large language models and games: A survey and roadmap," *IEEE Transactions on Games*, 2024.

[17] P. Sweetser, "Large language models and video games: A preliminary scoping review," in *Proceedings of the 6th ACM Conference on Conversational User Interfaces*, 2024, pp. 1–8.

[18] M. U. Nasir, S. James, and J. Togelius, "Word2world: Generating stories and worlds through large language models," *arXiv preprint arXiv:2405.06686*, 2024.

[19] M. U. Nasir and J. Togelius, "Practical pcg through large language models," in *Proceedings of the IEEE Conference on Games*, 2023.

[20] Z. Jiang, S. Earle, A. Khalifa, and J. Togelius, "Agentic pcg: Procedural content generation via tool-using llms," *Available at SSRN 6499021*, 2026.

[21] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A Field Guide to Genetic Programming*, 2008.

[22] J. R. Koza, *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science Stanford, CA, 1990, vol. 34.

[23] ——, "Hierarchical automatic function definition in genetic programming," in *Foundations of Genetic Algorithms*. Elsevier, 1993, vol. 2, pp. 297–318.

[24] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *arXiv preprint arXiv:2305.16291*, 2023.

Appendix

The following figures document the LLM call sites used by the evolutionary loop. Template variables such as `{domain_block}`, `{eval_feedback}`, and `{parent_program}` are filled at runtime.

Prompt 1: Mutation

Fires every generation with ∼75% probability (when crossover is not selected). All four conditions.

System Message

```
You are a Python level-generator mutation engine. Output only valid Python code.
```

For CL conditions, once the helper module has functions, this is extended with:

```
  You may use `from helper_loderunnertile_<uuid> import *` to access
  reusable utility functions discovered during this run.
```

User Prompt

```
You are mutating a Python level-generator function.

Task:
Produce ONE new Python ``generate(context_dict)`` function that differs
from the parent in a meaningful way.

Goal:
Create a candidate program that produces better-quality levels for the
target domain.  The function receives a context dict with keys
``"level"`` (2-D list of ints), ``"width"`` (int), and ``"height"`` (int).
It must return a 2-D list of ints of the same shape.

CRITICAL -- use the input level:
The function MUST start from ``context_dict["level"]`` -- the 2-D grid
provided as input.  Use it as the base grid to modify (e.g. clear
entities, reshape platforms, place new items).  Do NOT create a new empty
grid from scratch (e.g. ``[[0]*width for _ in range(height)]``).  A
typical first step is to copy the input level and strip entities you want
to re-place:
  level = [row[:] for row in context_dict["level"]]
  height = context_dict["height"]
  width = context_dict["width"]

You may use any Python standard-library or numpy code.  The function
will be ``exec()``-ed so all imports must be inside the function or at
module top-level.

Mutation strength: strong

Interpretation:
- light: tweak 1-2 numeric constants, add/remove a small code block
- medium: restructure a loop, change strategy for a subtask
- strong: rewrite larger sections but keep the overall generation approach

Do not:
- import or use ``sketchnl``, ``sketchnl.runtime``, or any ``rt.*`` functions
- output commentary, explanations, or markdown fences
- output anything other than valid Python source code
- define classes; only define a ``generate`` function (helper functions ok)
- hardcode level layouts: the function must use the input grid and
  algorithmic logic to produce varied outputs on every call
- ignore the input level: NEVER create a blank grid from scratch -- always
  start from ``context_dict["level"]`` and modify it
- change the function signature: it MUST be ``def generate(context_dict)``
- change the return type: MUST return a 2-D list of ints

Manipulate the 2-D tile grid directly using standard Python (loops, list
comprehensions, random, numpy, collections, etc.).  Refer to tiles by
their integer IDs as listed in the domain block below.

{domain_block}
{eval_feedback}
Parent program:
```python
{parent_program}
```
```

Fig. 6. Mutation prompt. The prompt specifies the `generate(context_dict)` contract, domain constraints, run memory, available helper APIs, mutation strength, and the parent program.

Prompt 2: Crossover

Fires with ∼25% probability when the archive has ≥2 agents. All four conditions.

System Message

```
You are a Python level-generator crossover engine. Output only valid Python code.
```

Same CL extension as mutation when applicable.

User Prompt

```
You are performing crossover between two Python level-generator functions.

Task:
Produce ONE child Python ‘‘generate(context_dict)‘‘ function that
combines ideas from both parents.

Goal:
Create a child that produces better-quality levels by combining the
strategies of both parents.  The function receives a context dict with
keys ‘‘"level"‘‘ (2-D list of ints), ‘‘"width"‘‘ (int), and
‘‘"height"‘‘ (int).  It must return a 2-D list of ints of the same shape.

CRITICAL -- use the input level:
The function MUST start from ‘‘context_dict["level"]‘‘ -- the 2-D grid
provided as input.  Use it as the base grid to modify (e.g. clear
entities, reshape platforms, place new items).  Do NOT create a new empty
grid from scratch (e.g. ‘‘[[0]*width for _ in range(height)]‘‘).  A
typical first step is to copy the input level and strip entities you want
to re-place:
  level = [row[:] for row in context_dict["level"]]
  height = context_dict["height"]
  width = context_dict["width"]

You may use any Python standard-library or numpy code.

Recombination strength: strong

Interpretation:
- light: take constants / parameters from both parents
- medium: combine strategies or code sections from both
- strong: larger recombination but keep functional correctness

Do not:
- import or use ‘‘sketchnl‘‘, ‘‘sketchnl.runtime‘‘, or any ‘‘rt.*‘‘ functions
- output commentary, explanations, or markdown fences
- output anything other than valid Python source code
- define classes; only define a ‘‘generate‘‘ function (helper functions ok)
- hardcode level layouts: the function must use the input grid and
  algorithmic logic to produce varied outputs on every call
- ignore the input level: NEVER create a blank grid from scratch -- always
  start from ‘‘context_dict["level"]‘‘ and modify it
- change the function signature: it MUST be ‘‘def generate(context_dict)‘‘
- change the return type: MUST return a 2-D list of ints

Manipulate the 2-D tile grid directly using standard Python (loops, list
comprehensions, random, numpy, collections, etc.).  Refer to tiles by
their integer IDs as listed in the domain block below.

{domain_block}
{eval_feedback}
Parent A:
‘‘‘python
{parent_a}
‘‘‘

Parent B:
‘‘‘python
{parent_b}
‘‘‘
```

Same {domain_block}, {eval_feedback}, and CL helper block as Prompt 1.

Fig. 7. Crossover prompt. The prompt receives two parent programs and asks the model to combine compatible mechanisms into a complete child program.

Prompt 3: Correction

Fires when a mutation/crossover output fails to compile or crashes during smoke-testing. Up to 3 retries. All conditions.

System Message

```
You are a Python syntax-correction engine.  Given a Python program and its
error traceback, output ONLY the corrected Python source code.  No
commentary, no markdown fences.  Do NOT import or use sketchnl,
sketchnl.runtime, or any rt.* functions.  Manipulate the tile grid
directly using standard Python.
```

User Prompt

```
The following Python program failed to compile or run.  Fix it.

Program:
```python
{code}
```

Error:
{error}
```

Fig. 8. Correction prompt. The prompt is used when a mutation or crossover output fails to compile or crashes during smoke testing. It receives the broken code and the error traceback.

Prompt 4: Memory Reflection

Fires after every generation in all four conditions (memory is enabled by default).

System Message

```
You are reflecting on one iteration of an evolutionary level-generator
search. Be concise and specific. Output markdown only -- no commentary
outside the requested format.
```

User Prompt

```
You are maintaining a memory file for an evolutionary level-generator search.
After each generation you must write a short structured entry summarising
what happened and update the global learnings.

Here is the memory so far (may be empty on the first generation):
---
{existing_memory}
---

This generation:
- Generation: {generation}
- Mechanism: {mechanism}
- Parent: {parent_id} (fitness {parent_fitness:.4f})
- Child fitness: {child_fitness:.4f} (delta {fitness_delta:+.4f})

Child evaluation:
{eval_feedback_text}

Child code:
```python
{child_code}
```

Write TWO sections of markdown (nothing else):

1. A generation entry with this exact format:

Gen {generation} ({mechanism} from {parent_id}, fitness {child_fitness:.4f})
- Parent: {parent_id} (fitness {parent_fitness:.4f})
- Tried: <1-sentence description of the key change vs parent>
- Eval: <copy the tier scores and FOCUS marker from the eval above>
- Failures: <top 3 failure modes with counts>
- Learned: <1-2 sentences on what this result teaches us>
- Code structure:
  1. <step 1>
  2. <step 2>
  ...

2. An updated Global Learnings section:

Global Learnings
- <bullet 1>
- <bullet 2>
...

Keep global learnings to 3-7 bullets. Update them based on the full
history -- drop stale insights, reinforce confirmed ones, add new ones.
```

{eval_feedback_text} is rendered from per-child analysis:

```
Parent evaluation (30 attempts):
  fitness (quality+diversity+valid%)/3: 0.4567
  mean quality: 0.7234
  intra-diversity: 0.3456
  valid levels (quality=1): 30%
  Quality tiers (each must reach 1.0 to unlock the next):
    Tier 1 -- entity counts: 1.00/1.00 (passed)
    Tier 2 -- exploration: 1.00/1.00 (passed)
    Tier 3 -- play stats: 0.85/1.00 <-- FOCUS HERE  (gold_collected=6/8, ...)
    Tier 4 -- decoration: 0.00/1.00 (blocked)
  Top failure modes:
    unreachable_gold: 12/30
    low_used_tiles: 8/30
  Goal: increase the percentage of valid levels (currently 30.0%)
```

Fig. 9. Memory reflection prompt. The prompt maintains a structured per-run memory with generation observations and global lessons for later variation calls.

Prompt 5: CL Helper Extraction

Fires every 5 generations starting at generation 8 for CL conditions only (`baseline` + CL, `gen` + CL). One call per high-fitness program in the window.

System Message

```
You are a code refactoring engine that extracts reusable primitives from
Python programs into a shared helper module.  Output ONLY valid Python
source code -- no commentary, no markdown fences.
```

User Prompt

```
Below is a Python level-generator program ({program_id}), followed by the
current contents of a shared helper module.

Your task: extract reusable DOMAIN UTILITIES from this program into the
shared helper module.  Think of the helper module as a curated API of
composable building blocks -- like a standard library for level generation.

The goal is to provide validation, repair, reachability, and grid-primitive
functions that ANY generation strategy can benefit from.  Do NOT extract
generation strategies or high-level orchestration logic.

What to extract (priority order):

  TIER 1 -- Validation & repair (highest priority):
    - Entity reachability checking and relocation
    - Entity count normalization (ensure exactly N of tile type)

  TIER 2 -- Domain-aware primitives:
    - Grid search utilities
    - Complete nested function definitions inside generate() that implement
      a domain utility.  Promote them to module-level with grid/height/width
      as explicit arguments.

  TIER 3 -- Generic algorithms (only if non-trivial, 15+ lines):
    - Pathfinding (A*, Dijkstra) with domain-specific movement rules
    - Constraint solvers that validate placement decisions

Good examples (EXTRACT these):
  - ensure_one_player(level, height, width)
  - grounded_empty_cells(grid, height, width)
  - normalize_entity_count(level, height, width, tile, target)

BAD examples (NEVER extract):
  - The entire generate() body or its top-level orchestration logic
  - clear_tile(grid, tile) -- trivial single loop
  - solidify_bottom_row(grid, width) -- trivial single row fill

Each function should be 5-40 lines of body code (excluding docstring).

Already extracted in this batch (do NOT duplicate these):
  {already_extracted}

Rules:
- Output the COMPLETE updated helper file (not just the new additions).
- PRESERVE every existing function and constant -- never remove or rename.
- Append new functions at the end of the file.
- Each new function MUST have a Google-style docstring with:
  (1) a one-line summary, (2) an Args section listing every parameter
  with its type, (3) a Returns section, (4) an Example section.
- Use Python type hints in the function signature.
- Keep the module self-contained: only stdlib imports -- no numpy.
- If no suitable domain utilities exist, return the file UNCHANGED.

--- Program ({program_id}) ---
‘‘‘python
{program_code}
‘‘‘

--- Current helper module ({helper_name}) ---
‘‘‘python
{helper_contents}
‘‘‘

Output the complete updated helper module:
```

Fig. 10. CAD helper extraction prompt. Starting at generation 8 and every five generations thereafter, the prompt extracts reusable domain utilities from programs in the top fitness quartile.

Prompt 6: CL Helper Correction

Fires when a CL-extracted helper module fails to compile. Up to 2 retries. CL conditions only.

System Message

```
You are a Python syntax-correction engine. Given a Python helper module
and its compilation error, output ONLY the corrected complete Python
source code. No commentary, no markdown fences.
```

User Prompt

```
The following Python helper module failed to compile. Fix the error and
output the COMPLETE corrected module.

Module:
```python
{code}
```

Error:
{error}
```

Fig. 11. CAD helper correction prompt. The prompt receives the helper module and its compile or smoke-test error.

Prompt 7: Refactor

Fires with 25% probability after mutation/crossover when CL is active and the helper module has functions. CL conditions only.

System Message

```
You are a Python level-generator refactoring engine.  Your goal is to
simplify a generate() function by replacing inline code with calls to
tested helper functions from the helper module.  The refactored program
MUST produce identical output to the original given the same input and
random seed -- do not change any constants, random ranges, loop bounds,
or algorithmic logic.  Output only valid Python code -- no commentary,
no markdown fences.
```

User Prompt

```
Below is a level-generator program and a helper module with tested,
reusable functions.  Your task: refactor the program to USE the helper
functions wherever they can replace equivalent inline logic.

Rules:
- Replace inline code with helper calls where the helper does exactly
  the same thing.  Do NOT change constants, random ranges, or loop
  bounds.  Prefer shorter, cleaner code.
- Do NOT change the overall generation strategy or algorithm.
- Do NOT remove logic that has no helper equivalent -- leave it as-is.
- Use ``from <helper_module> import *`` at the top and call functions
  directly.
- The refactored program must still define ``def generate(context_dict)``
  and return a valid level grid.
- If no helpers are applicable, return the program unchanged.

{eval_feedback_block}

Parent program:
```python
{parent}
```

{helper_block}

Output the complete refactored program:
```

Fig. 12. Refactoring prompt. The prompt rewrites a generator to replace duplicated inline logic with calls to tested helper functions.

Prompt 8: Missing Helper Scaffold

Fires when a generated program imports a function that does not exist in the CL helper module. CL conditions only.

System Message

```
You are a Python helper-function generator.  Given the name of a missing
function, the program that calls it, and the current helper module, output
ONLY the new function definition (with a Google-style docstring including
an Example section).  No commentary, no markdown fences, no duplicate of
existing code -- just the new ``def``.
```

User Prompt

```
The following program tried to call ``{helper_name}.{func_name}`` but
that function does not exist in the helper module yet.

Program that uses it:
```python
{code}
```

Current helper module ({helper_name}):
```python
{helper_source}
```

Write ONLY the missing function ``{func_name}`` so it can be appended to
the helper module. Follow the same style (type hints, Google docstring
with Args/Returns/Example sections). The function must be self-contained
using only stdlib imports already in the helper module.
```

Fig. 13. Missing-helper scaffold prompt. The prompt is used when a CAD program imports a helper that does not yet exist in its run-specific module.