Work supported by

UKRI Engineering and Physical Sciences Research Council

ActionLearn

A White Paper on Full-Stack, Real-World Reinforcement Learning

January 2026

Carlos Purves
PhD Candidate
University of Cambridge

Preface

In the years since I started my PhD work, Reinforcement Learning in the 'Real World', the relationship between AI and the real world has evolved continuously. Nowadays, for better or worse, ostensibly general-purpose AI finds a role in almost all parts of our lives—a trend destined to continue as I write at the start of 2026. Despite this, the challenges of deploying complex applications into resource-critical systems persist, as do the risks involved in critical scenarios where a person's life chances, safety, or life might be at stake.

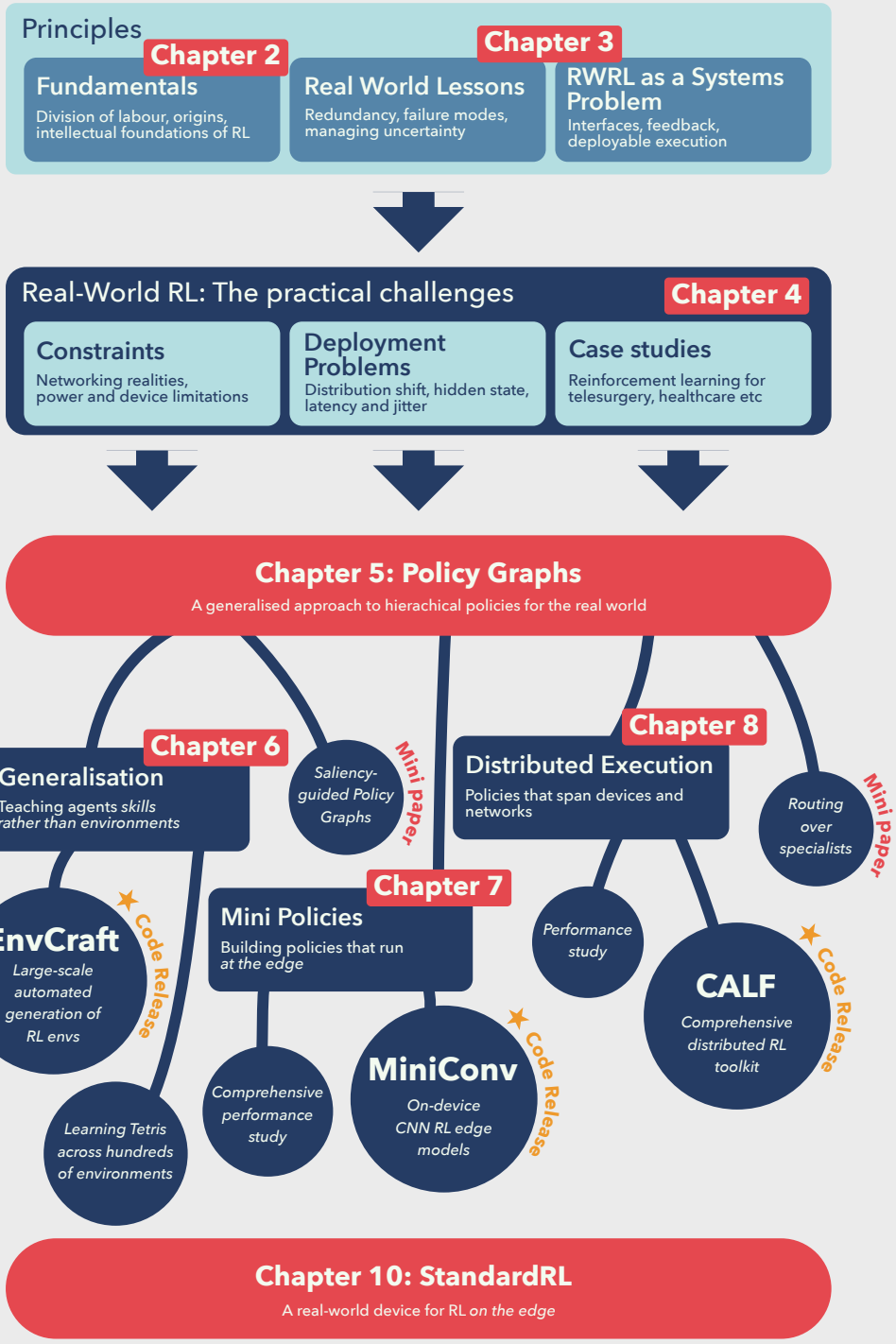
In the document that follows, we dig deeply into the foundations of Reinforcement Learning, walking from theoretical fundamentals through network protocols, containerisation, OpenGL, to high-level systems and finally to the design and use of a real-world hardware device. For those who prefer an **a la carte** experience, I provide a thesis map at the beginning.

Carlos Purves

Department of Computer Science and Technology

University of Cambridge

Thesis Map


Principles
Chapter 2
Fundamentals
Division of labour, origins, intellectual foundations of RL
Chapter 3
Real World Lessons
Redundancy, failure modes, managing uncertainty
RWRL as a Systems Problem
Interfaces, feedback, deployable execution
Real-World RL: The practical challenges
Chapter 4
Constraints
Networking realities, power and device limitations
Deployment Problems
Distribution shift, hidden state, latency and jitter
Case studies
Reinforcement learning for telesurgery, healthcare etc
Chapter 5: Policy Graphs
A generalised approach to hierachical policies for the real world
Chapter 6
Generalisation
Teaching agents skills rather than environments
Saliency-guided Policy Graphs
Mini paper
Chapter 8
Distributed Execution
Policies that span devices and networks
Routing over specialists
Mini paper
Chapter 7
Mini Policies
Building policies that run at the edge
EnvCraft
Large-scale automated generation of RL envs
Code Release
Learning Tetris across hundreds of environments
Comprehensive performance study
MiniConv
On-device CNN RL edge models
Code Release
Performance study
CALF
Comprehensive distributed RL toolkit
Code Release
Chapter 10: StandardRL
A real-world device for RL on the edge

Contents

Chapter 1

Introduction

1.1 Introduction

Reinforcement learning offers a route to autonomous decision-making through environmental interaction. Yet the path from simulated Atari games to real-world deployment confronts fundamental obstacles: policies overfit to narrow training distributions, learned behaviours lack interpretability, communication latency undermines reactive control, and edge devices impose severe computational constraints. This thesis addresses these challenges by asking a concrete design question: what happens if reinforcement learning borrows its architecture from real systems that already operate reliably under constraint—the A320's flight computers, the French power grid's layered control, and, at a still more abstract level, the division of labour in pin factories?

Chapter 2 (*Principles*) traces automation from first principles. Adam Smith observed that dividing pin-making into eighteen operations enabled ten workers to produce 48,000 pins daily—a 240-fold improvement over craftwork. Dopamine neuroscience reveals how phasic spikes encode reward prediction error, the brain's mechanism for reinforcing successful actions and chunking them into reusable routines. These threads converge: specialisation improves productivity, reward signals drive learning, and modular organisation enables both.

Chapter 3 (*Lessons*) examines how engineered systems achieve reliability through redundancy, sensor fusion, and failsafes. The A320 distributes responsibility across dedicated computers (ELACs for pitch and roll, SECs for spoilers and backup, FCGCs for autopilot); the power grid coordinates IEDs at substations with SCADA at national scale; the Kangduo surgical robot maintains sub-300ms latency through dual-console handover. These systems embody principles that learned policies must inherit: constrained transitions prevent mode confusion, commitment bounds enable predictable execution, explicit delegation provides accountability.

Chapter 4 (*Works*) surveys real-world RL deployments. Across sepsis treatment, sur-

gical robotics, and autonomous driving, a consistent pattern emerges: policies overfit to training conditions, cannot explain their decisions, cannot guarantee bounded execution, and fail under the computational constraints of deployment hardware. These four gaps organise the contributions that follow.

Chapter 5 (*Effects*) introduces policy graphs, a formalism distilling real-world architectural patterns into reinforcement learning. A directed graph $G = (V, E)$ defines callable policy units with hard routing—exactly one unit active at any moment—providing accountability (call traces identify responsible units), conditional computation (only active unit incurs cost), and distributed execution (units map to heterogeneous hardware). System 1 impulses execute on low-power edge devices near actuators; System 2 reasoning runs on remote GPU clusters. Commitment bounds $(k_{\min}, k_{\max})$ prevent unstable switching whilst ensuring progress. The chapter then studies two construction routes: a saliency-guided synthesis path that derives specialists from a teacher policy in controlled MiniGrid settings, and a hard-routing study over fixed specialists in deployment-motivated environments such as BrowserEnv, ViZDoom, and Procgen.

Chapter 6 (*Generalisations*) addresses benchmark scarcity. Traditional RL benchmarks comprise dozens of manually designed tasks; distinguishing generalisation from memorisation requires diverse environment families. EnvCraft generates thousands of validated Gymnasium environments from natural-language concepts through a multi-stage pipeline combining a code-generation LLM (a lightweight model for brief generation, a larger model for implementation), automated testing, and agent-based validation. Cross-validation experiments on procedurally generated Tetris variants provide within-family evidence that training diversity can improve performance on held-out variants.

Chapter 7 (*Models*) realises an edge-oriented split-policy deployment path. MiniConv provides compact convolutional encoders that compile to OpenGL fragment shaders for broad embedded GPU support. A split-policy architecture places lightweight encoders on-device (Raspberry Pi Zero 2 W, NVIDIA Jetson Nano), extracting compact features transmitted to remote policy heads. This reduces decision latency in bandwidth-limited settings and lowers server-side compute per request whilst remaining competitive with the Stable-Baselines3 Full-CNN baseline in the reported fixed-seed pixel-observation experiments.

Chapter 8 (*Systems*) extends the thesis to network-aware distributed execution. CALF treats environments and policy units as networked services, injecting latency, jitter, and packet loss during training. Without network-aware training, a CartPole policy loses over 80% of its return under degraded Wi-Fi; the same policy trained under CALF degrades by only 21%, a roughly four-fold reduction in the sim-to-real gap. Small hierarchical deployments then illustrate how time-critical units can remain local whilst higher-level coordination runs remotely.

Chapter 9 (*Realisations*) sketches an initial physical-device pathway: a USB-C de-

vice built around a Raspberry Pi Zero 2 W that captures DisplayPort video from a host computer, runs MiniConv inference locally, and returns HID actions over the same connection—placing a trained policy graph inside an unmodified host machine's input chain.

Chapter 10 (*Endings*) synthesises the contributions and returns the thesis argument to its origins. Smith's pin-factory productivity claim is revisited alongside Diderot's correction, and Plato's cave is extended to network delay: agents cannot escape incomplete observations, but network-aware training teaches them to navigate the shadows they perceive.

Taken together, the chapters argue that real-world deployment of reinforcement learning requires more than algorithmic performance on benchmarks. It demands operational semantics that provide interpretability and bounded execution, training infrastructure that tests generalisation beyond narrow distributions, and system architectures that distribute computation across heterogeneous hardware whilst maintaining accountability under communication constraints. By grounding modular RL in the architectural patterns of engineered systems—from pin factories to flight computers—this thesis offers a principled pathway from simulation to deployment.

1.2 A Note on Reproducibility

This thesis prioritises replicable work. Non-replicable publications attract more citations than reproducible ones [1], likely because reviewers "apply lower standards regarding reproducibility" for "more interesting" findings; this work does not aspire to that trade-off. Original contributions provide code at publication or shortly after. Where practicable, a  symbol indicates browser-based reproducibility support: the associated QR code links to an executable artefact or live validation page for the claim in question. Server capacity is committed for one year post-publication, with some services potentially remaining available for longer.

Chapter 2

Principles

Adam Smith, in his seminal work *The Wealth of Nations* (1776), chose pin-making as his primary exemplar of *the division of labour*. Smith described how a worker alone, even when employing "his utmost industry" could make "one pin in a day", but certainly, Smith posed, he could "not make twenty". By dividing labour across ten workers, each with their own speciality and familiarity with certain machines, forty eight thousand pins could be produced in a day. "The greatest improvement in the productive powers of labour", Smith surmised, "seem to have been the effects of the division of labour".

This thesis asks how a simple and enduring idea, the division of labour, can help make reinforcement learning systems more deployable in the real world. It begins with the first principles of automation: its history, its philosophical foundations, and the path by which reinforcement learning reached its current form. It then considers the present state of the art, the reasons existing approaches still struggle in real deployment settings, and the methods introduced here to address those gaps.

We begin with *labour*.

2.1 Labour

In Plato's *Phaedrus* (265E), Socrates describes the process of carving nature "without trying to shatter a single part by going about it like a bad butcher ... on the basis of Forms [and] according to its natural joints". Dividing the labour of pin-making, in the style of Smith, similarly involves carving up the task by its natural joints.

In the case of pin-making, Smith states that eighteen "distinct operations" were used in producing pins, with some factories employing different people for each and others where employees performed two or three tasks each. Whether Smith truly visited any pin factories to come to these conclusions is unknown, in part due to his request that his contemporaneous notes be burned before his death. Because of the details he mentions—including his belief that specifically eighteen steps are used to produce pins—it is

likely that he was borrowing heavily from Denis Diderot's *Encyclopédie*.

The *Encyclopédie* includes an article detailing the eighteen purported steps used by Parisian pin makers, written by Alexandre Delaire. An overview of the steps is shown in Figure 2.1. Delaire, a literature specialist, was picked to obfuscate the plagiarism of the previous pin-making article after Jesuits accused Diderot of copying twenty-two articles from the French Academy of Sciences, including the original pin-making article. The separation of pin-making into eighteen operations appears to have been a literary invention of Delaire to avoid the claim of plagiarism, with original sources suggesting that there may have been closer to six distinct skills used within a factory. In addition to revealing the fabrication of the number of skills, original sources also raise questions about the methodology and conclusions detailed by Smith [2].

Whatever the specific truth of Smith's claims, it is clear that the separation of skills along natural joints can significantly improve productivity. The best metrics for productivity, and the best ways to incentivise work, are the questions to which we turn next.

2.2 Reward

Since antiquity, philosophers recognised pleasure and pain as behavioural motivators. From Epicurus's observation that pleasure is "our first and kindred good" [3] to Jeremy Bentham's formalisation of this insight in *Introduction to the Principles of Morals and Legislation* as the 'felicific calculus'—an approach to quantifying utility in units of pleasure (*hedons*) and pain (*dolors*)—the same basic premise has structured thinking about motivation across centuries. This framework acknowledged that actions might yield different utility across time and context.

As the field of behaviourism developed at the end of the 19th century, Edward Thorndike proposed his *Law of Effect*, formalising notions of reward for the first time. The Law of Effect states that behaviours followed by a reward are more likely to be repeated in the future. Ivan Pavlov demonstrated 'classical conditioning' through experiments with dogs, showing that involuntary responses could be conditioned to neutral stimuli. B.F. Skinner distinguished this 'respondent behaviour' from 'operant behaviour', in which animals learn to associate conscious actions with rewards.

In *Behavior of Organisms* (1938), Skinner formally defined reinforcement as "the presentation of a certain kind of stimulus in a temporal relation with either a stimulus or response". He introduced 'intermittent reward', demonstrating that whilst continuous reward enables faster learning, intermittent reward produces more robust behaviour less susceptible to 'extinction' when reinforcement ceases. Skinner also proposed 'reward shaping', in which difficult tasks are decomposed into smaller, more achievable units, each eliciting incremental reward.

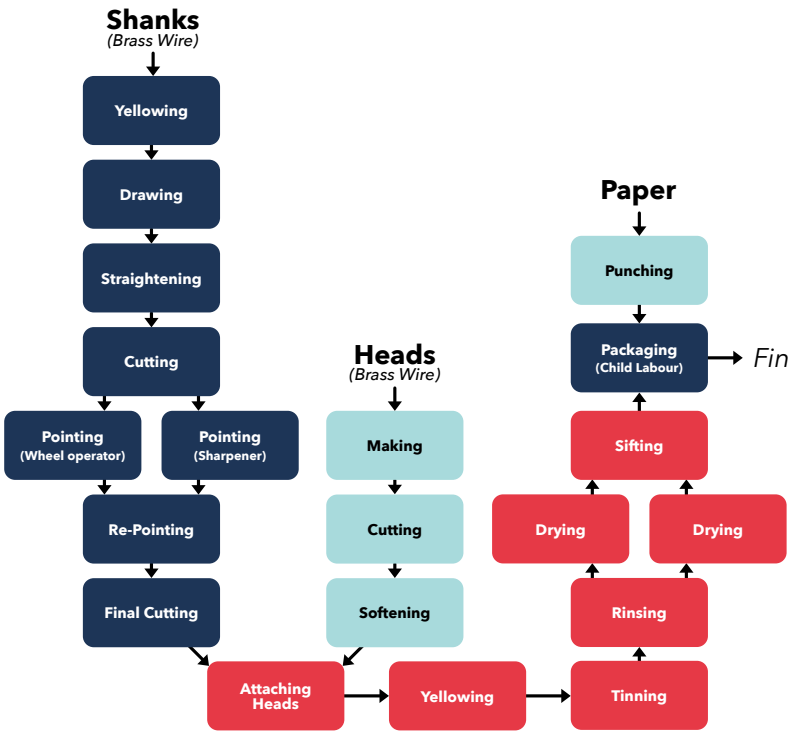


Figure 2.1: The steps involved in the process of pin-making, as written by Alexandre Delaire in *Encyclopédie*, translated from original French and heavily simplified. The original source text is given in Appendix A. Brass wire is used to form shanks and heads, which are then combined and packaged with paper. The processes of pointing and dyeing are described by Delaire as using two workers each.

Clark Leonard Hull articulated how a reinforcing stimulus could take the form of a 'reduction of a drive'. A drive, such as hunger, is alleviated by eating food; thus the provision of food to a hungry dog has reinforcing effect.

Parallel to the works of Thorndike, Pavlov, Skinner and Hull, neuroscience would establish the biological substrate underlying these behavioural principles. The works of Kathleen Montagu and Arvid Carlsson identified dopamine as the neurotransmitter regulating reward processing and behaviour. Crucially, neuroscientific investigation revealed that dopamine does not simply signal the presence of reward; rather, it encodes **reward prediction error**—the difference between received and expected reward. When an outcome is better than anticipated, dopamine activity spikes; when an outcome disappoints, it falls below baseline. This mechanism provides a biological precedent for temporal-difference learning, in which an agent learns to predict future reward and updates those predictions based on observed discrepancies between expectation and outcome. The bridge from Skinner's reinforcing stimulus to the update rules of modern reinforcement learning runs directly through this neurobiological discovery.

This research forms an important foundation for reinforcement learning, where an artificial notion of 'reward' is used to guide the training of an artificial system. To move from neuroscience to artificial intelligence, it helps first to understand human behaviour in terms of *systems*.

2.3 Systems I

Human behaviour ranges from pushing buttons to complex gymnastics. So far, we have considered the ways in which behaviour is learned: classical conditioning to learn simple reflexive tasks and operant conditioning for more complex incentive-driven behaviour. We have also discussed the role of dopamine as the biological substrate of reward prediction error. However, it is hard to conceive of the ways in which, say, learning to keep balance on two feet could relate to learning a gymnastic routine. One thing we can say for sure is that many of the things we learn through conditioning can eventually be executed without thinking carefully about them. Balancing on a bike might take work at first, but eventually it becomes easier to balance with less thinking rather than more. There is some mechanism that takes smaller actions, such as the subtle movements involved in balancing, and groups them in a way that makes their execution feel automatic.

Cognitive dichotomies distinguishing reflexive from deliberative processes recur throughout intellectual history, from Descartes' mind-body distinction to modern theories of dual-process cognition. Herbert Simon formalised this in his *Theory of Bounded Rationality* (1957), arguing that human decision-making operates within cognitive and informational constraints. He identified two principal modes of thought: heuristic-driven processes, which rely on rules of thumb and mental shortcuts to make quick decisions, and rational

processes, which require deliberate, logical analysis.

In more recent years, the notion of a systematic separation of aspects of human cognition has been brought to popular fame through Kahneman's *Thinking, Fast and Slow* (2011). The System 1/System 2 dichotomy was introduced by Stanovich and West [4] and popularised by Kahneman, who writes that System 1 "operates automatically and quickly, with little or no effort and no sense of voluntary control" whereas System 2 is used for "effortful mental activities" and is associated with "agency, choice, and concentration". He describes the relationship between System 1 and System 2 as a "division of labor".

This cognitive dichotomy informs our approach to distributing computation across heterogeneous hardware: reactive control on resource-constrained edge devices (analogous to System 1), and deliberative planning on remote servers with greater computational capacity (analogous to System 2).

We turn now to the history of automation that made these computational frameworks necessary.

2.4 Automation

Automation has fascinated thinkers throughout history. In Plato's *Meno* (97d), Socrates describes Daedalus' mythical living statues as "play[ing] truant and run[ing] away", "if they are not fastened". In his work *Politics* (Book 1), Aristotle considers the set of 'tools' that exist as comprising two parts: 'living' and 'lifeless'. Living tools include human assistants and lifeless ones include implements like the rudder of a ship. He supposes that "if every tool could perform its own work when ordered, or by seeing what to do in advance, like the statues of Daedalus in the story", then "master-craftsmen would have no need of assistants".

The *Mechanical Turk*, constructed in 1770, appeared to play chess autonomously, defeating opponents including Benjamin Franklin; Napoleon Bonaparte famously lost to it in 1809. In reality, it was an elaborate fraud: expert chess players concealed themselves within the desk, operating the mannequin from hidden compartments using an ingenious system of sliding seats and mirrors [5]. Despite its deception, the Turk profoundly influenced subsequent work in automation, including that of Charles Babbage, who lost to it twice during its 1819 European tour.

Babbage, lamenting errors in hand-calculated logarithm tables, conceived the *Difference Engine* (for automating calculations) and later the *Analytical Engine* (for general computation). His collaborator Ada Lovelace wrote what is widely acknowledged as the 'first ever algorithm' for the Analytical Engine. Babbage, likely influenced by the Mechanical Turk, believed his Analytical Machine could play chess competently.

Leonardo Torres Quevedo was a big admirer of Babbage's work. In his 1914 essay *Ensayos sobre automática*, he credits Babbage's "mechanical genius" (genio mecánico) and

describes him as a "distinguished mathematician" (matemático distinguido). He wished to extend the theoretical work of Babbage, who was never able to finish constructing his theorised computer. Torres was extremely optimistic about automation. His work serves, for our purposes, as the conclusive bridge from the first principles of automation to modern day algorithmic artificial intelligence.

In a supplement to the November 1915 issue of *Scientific American*, Torres' "Remarkable Automatic Devices" are profiled, alongside the claim that Torres "Would Substitute Machinery for the Human Mind". Indeed, what follows reads strikingly like an early manifesto for modern artificial intelligence[1]:

> When it comes to an apparatus in which the number of combinations makes a very complex system, analogous in a small degree to what goes on in the human brain, it is not generally admitted that a practical device is possible. On the contrary, M. Torres claims that he can make an automatic machine which will "decide" from among a great number of possible movements to be made, and he conceives such devices, which if properly carried out, would produce some astonishing results. Interesting even in theory, the subject becomes of great practical utility, especially in the present progress of the industries, it being characterised, in fact, by the continual substitution of machine for man; and he wishes to prove that there is scarcely any limit to which automatic apparatus may not be applied, and that at least in theory, most or all of the operations of a large establishment could be done by machine, even those which are supposed to need the intervention of a considerable intellectual capacity.

The single most notable "remarkable automatic device" in the Scientific American profile of Torres was *El Ajedrecista*: one of the first chess-playing automata to operate through genuinely algorithmic means rather than hidden human operators. Playing a simplified endgame (king and rook versus king), it demonstrated that machines could exhibit strategic reasoning through programmed rules. The automaton would later famously play against chess Grandmaster Savielly Tartakower in 1951.

Torres, in his 1914 essay, explains the importance of machines having what he calls *discernment* (discernimiento). "It is necessary", he says "and this is the main object of Automation" (y éste es el principal objeto de la Automática), that they can choose the correct action by "taking into account the impressions they receive, and also, sometimes, those they have received previously". He points towards a distinction between the types of machines that people generally believe possible. On one hand, machines which respond continuously to stimulus input are agreed to be easy to make, whereas those which

[1] In a now plainly discreditable sign of the period, the profile appears shortly after material endorsing eugenic research.

"[weigh] the circumstances surrounding [them] ... in determining ... actions" (pese las circunstancias que le rodean) are "generally thought" to be "[achievable] in very simple cases". He claims that "this distinction is worthless" (esta distinción carece de valor), and that "it is always possible to build an automaton whose acts, all of them, depend on certain more or less numerous circumstances, obeying rules that can be arbitrarily imposed at the time of construction".

The 1951 Festival of Britain featured *Nimrod*[2], which played the game nim and drew such crowds during its European tour that special police were required for crowd control.

In 1948, Alan Turing and David Champernowne developed TuroChamp[3], a powerful chess-playing algorithm. Due to its age, there was an awkward caveat to the algorithm; no 'computer' existed to run it. The computation for each move had to be carried out with pen and paper.

Amongst Turing's several questions about what it might mean to "make a machine to play chess", the most prescient was whether a machine could "improve its play, game by game, profiting from its experience"—question four of six, and the one he could not yet answer with confidence. He detailed instead an algorithm for question three: a machine that would indicate a passably good legal move. The algorithm operates by assigning each position a *value* derived from material and positional considerations, then selecting the move leading to the highest-valued position reachable within a limited search depth. The conceptual move from Torres's *discernimiento* to Turing's position *value* is the crucial one: it establishes that a machine can encode preferences over states as numerical quantities, and act so as to maximise them.

Across these episodes, automation moves from myth and spectacle towards useful computation. The Mechanical Turk traded on deception, but it still helped sustain public fascination with machine intelligence; Babbage, Torres, and Turing then redirected that fascination towards genuine mechanism and calculation. Turing's notion of *value* marks an especially important step towards what we now call reinforcement learning: behaviour guided by numerical preferences over future states. The developments that made this possible—feedback control, adaptive systems, and ultimately learning from interaction—emerged during the mid-twentieth century under the banner of cybernetics.

2.5 Heuristics

Three recurring heuristics structured how automation progressed through these episodes, and they recur throughout the RL methods that follow. *Bootstrapping* is the process of using something to improve itself: an initial capability becomes the basis for ac-

[2] *Not* the 19th-century self-styled prophet Nimrod Murphree, whose claims to unaided flight did not pan out.

[3] A portmanteau of their surnames.

quiring a more refined one, without external supervision. The term derives from the nineteenth-century expression for lifting oneself off the ground by pulling on one's own bootstraps—an impossibility in the physical sense, yet in machine learning it describes a genuinely productive loop, from temporal-difference value estimation to self-play. *Evolution* operates by iterative selection across a population: variation is introduced, fitness is evaluated, and successful variants propagate. In reinforcement learning this manifests in population-based training and neuroevolution, where diversity guards against premature convergence. *Co-evolution* extends this by making the selection pressure itself adaptive: as predator and prey evolve in mutual response, competing or co-operating agents drive one another's development. Self-play and adversarial curriculum generation are its direct expressions in deep RL. These three heuristics are named here so that they can be recognised when they reappear.

2.6 How to Train Your Machine

The mid-20th century saw the emergence of cybernetics, a discipline that formalised the role of feedback in control systems. Influenced by biological and neurological models, cybernetic approaches emphasise homeostatic regulation and adaptive response mechanisms. This was demonstrated by wartime innovations such as torpedo guidance systems and the *Homeostat*, an early self-regulating machine developed by W. Ross Ashby. As electronic computing matured, analogue control systems were widely adopted in industrial processes and aerospace applications, facilitating real-time automation. The transition from analogue to digital control in the 1960s and 1970s further improved precision, enabling the development of programmable controllers for manufacturing, as well as early forms of computerised decision-making in robotics and avionics.

Rule-based expert systems and fuzzy logic controllers extended this automation further, but remained dependent on hand-engineered knowledge and could not learn.

Reinforcement learning emerged as a response to these limitations, providing a framework in which control policies are learned through interaction with an environment rather than being explicitly programmed. Rooted in behavioural psychology and dynamic programming, RL enables machines to optimise decision-making by maximising cumulative rewards over time. This approach is particularly effective in scenarios where system dynamics are complex or only partially known, making it well-suited to robotics, adaptive automation, and real-time decision systems.

2.7 Deep Learning Foundations

The control systems described in the previous section—from cybernetic feedback loops to fuzzy logic controllers—rely on explicit modelling of system dynamics. Whilst effec-

tive in well-understood domains, these approaches struggle when confronted with high-dimensional observations. A robot navigating an indoor environment receives camera images: even a modest 84×84 pixel RGB frame corresponds to a 21,168-dimensional input. Enumerating rules or value tables at this scale becomes impractical. Real-world reinforcement learning demands *function approximation*—learning to generalise from observed states to unobserved ones.

Deep neural networks [6] provide the expressive capacity required. Stacked layers of parameterised transformations learn hierarchical representations: early layers detect edges and textures; later layers compose these into semantically meaningful features. Gradient-based training—specifically *backpropagation* [7] with adaptive optimisers such as Adam [8]—makes this practical at scale.

For visual observations, *convolutional neural networks* (CNNs) [9] are particularly well-suited. Rather than treating an image as a flat vector, convolutional layers apply learned filters that slide across the spatial extent of the image. This *parameter sharing* drastically reduces the number of trainable weights, and *local connectivity* reflects the natural structure of images: edges, textures and objects are spatially localised, and a useful detector for an edge in one region of the image is equally useful in another. Stacked convolutional layers followed by fully-connected output heads form the backbone of visual RL architectures and are used extensively in this thesis—most directly in Chapter 7, which develops compact CNN encoders for deployment on resource-constrained edge hardware.

Combining neural networks with reinforcement learning introduces training challenges that do not arise in supervised settings. Reinforcement learning generates data through interaction: consecutive observations are temporally correlated, and the data distribution shifts as the policy improves. The *deadly triad* [10]—combining function approximation, bootstrapping from value estimates, and off-policy learning—creates instability that naive gradient descent cannot resolve. Section 2.8.4 describes how Deep Q-Networks resolved these instabilities and established deep reinforcement learning as a practical methodology. The chapters that follow build throughout on the foundations introduced there.

2.8 Reinforcement Learning

We now reach the formal conceptual introduction of modern reinforcement learning. We will build the framework systematically, starting from first principles and progressively introducing the extensions—deep function approximation, policy gradient methods, hierarchical decomposition—that underpin the contributions in this thesis.

2.8.1 The World as Will and Environment

Let us consider Turing's third question about chess again:

> Could one make a machine which would play a **reasonably good game of chess**, i.e. which, confronted with an ordinary (that is, not particularly unusual) **chess position**, would after two or three minutes of calculation, indicate a passably good legal move? (Emphasis added.)

In order for a machine to make a "passably good legal move", it needs some way to interpret the "chess position". For Turing's chess algorithm, this state of play is represented by a full comprehension of the chess board. This is part of the reason for Turing's algorithm being possible to compute only on pen-and-paper. The computers that existed[4] at the time were too restrictive to interpret such a complex state and carry out the necessary number of computations. For Torres, automation should make moves by "taking into account the impressions they receive, and also, sometimes, those they have received previously", raising the possibility of distilling the state which the machine receives only down to its most relevant components.

Here, we will introduce the abstract principal paradigm for RL through its three components:

- The agent: the entity that carries out actions
- The environment: a representation of the world on which the agent acts
- The policy: the rulebook that the agent follows when deciding which action to take next in the environment

RL is the process by which the policy is learned and it is carried out by recording details of interactions that occur between the agent and the environment. The RL paradigm is *abstract*: both the environment and the agent conform to specific constraints of form which distinguish them from the real-world problem they represent. To formalise this, four key assumptions are made:

1. Time is composed of discrete 'steps', rather than being continuous
2. The agent can take actions at each step, but not between steps
3. Positive behaviour is indicated by the environment in the form of a numerical *reward*, although no constraints are made on the regularity or scale of the reward, just that it must eventually be provided

[4]The Ferranti Mark I (1951), an improved version of the Manchester Mark I, was the first commercially available general-purpose computer. Turing attempted to run his chess algorithm on the Ferranti but was unsuccessful in his lifetime.

4. The state of the environment at each step is sufficient to choose the optimal action

It is important to note that, for any real-world problem, there can be multiple different environments which suitably represent it, as well as many which appear to represent it, but which fail to adhere properly to the assumptions above. Assumption 4 is sometimes called the *Markov assumption*: the state must be sufficient to choose the optimal next action, so any information critical to the decision must be included in the state representation. We formalise this as a *Markov Decision Process* (MDP).

2.8.2 Markov Decision Processes

A Markov Decision Process (MDP) is a structure $\text{MDP}(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ where:

- $\mathcal{S}$ is a set of states.
- $\mathcal{A}$ is a set of actions.
- $\mathcal{T}(s'|s,a) = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a)$: The probability of a transition to state s' given current state s and action a.
- $\mathcal{R}(s,a,s') = \mathbb{E}(r_t|s_t = s, a_t = a, s_{t+1} = s')$: The expected reward gained when the system transitions from state s to s'.

MDPs exhibit the *Markov Property*.

Property 1 (Markov) *For any* $\text{MDP}(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ *and any* t *having* $a_{t-1}, \ldots, a_0 \in \mathcal{A}$ *and* $s_t, \ldots, s_0 \in \mathcal{S}$*:*

$$\mathbb{P}(s_t = s|\underbrace{[a_{t-1}, \ldots, a_0]}_{\textit{previous actions}}, \underbrace{[s_{t-1}, \ldots, s_0]}_{\textit{previous states}}) = \mathbb{P}(s_t = s|s_{t-1}, a_{t-1}) = \mathcal{T}(s_t|s_{t-1}, a_{t-1}). \tag{2.1}$$

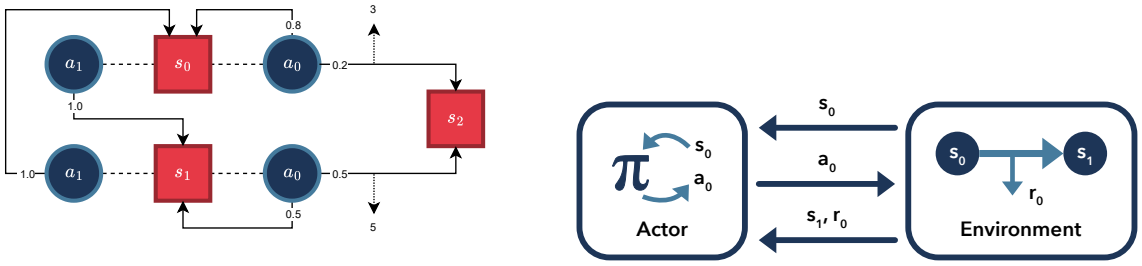
There may also be two sets S_{start} and S_{term} having:

$$\forall s \in \mathcal{S}\ \forall s' \in S_{\text{start}}\ \forall a \in \mathcal{A} \quad \mathcal{T}(s'|s,a) = 0, \tag{2.2}$$

$$\forall s \in \mathcal{S}\ \forall s' \in S_{\text{term}}\ \forall a \in \mathcal{A} \quad s \neq s' \implies \mathcal{T}(s|s',a) = 0. \tag{2.3}$$

A set of transitions from a state in S_{start} to one in S_{term} constitutes an *episode*. An example MDP is shown in Figure 2.2a and the view of interactions with the system of the agent is shown in Figure 2.2b.

If an action a causes a transition from a state s to s', then the tuple (s,a,s') is described as an *experience*. A *trajectory* (of length n) is an ordered set of experiences



(a) A simple MDP structure with three states. In this case, s_0 is a start state and s_2 is a terminating state.

(b) The way that an agent interacts with an environment. Each action gives the agent some reward and changes the environment's state.

Figure 2.2: *Left:* an MDP with three states—of which s_2 is a terminating state—and two actions. The action a_1 acts as a toggle between states s_0 and s_1 (and can be taken whilst in either of them). Taking a_0 in either state causes a transition to s_2 with a certain probability. Rewards are represented by arrows pointing from state transitions. *Right:* the canonical form of an MDP, in which an agent performs an action a_t on an environment causing it to undergo an internal state transition $s_t \to s_{t+1}$. This transition yields a reward of r_t.

that describes a series of actions taken by the agent to move the environment from some state s_0 to a state s_n:

$$\tau = \{(s_0, a_0, s_1), (s_1, a_1, s_2), \ldots, (s_{n-1}, a_{n-1}, s_n)\}. \tag{2.4}$$

Note that if $s_n = s_{\text{term}}$, then τ describes an *episode*.

2.8.3 The Trial (and Error)

The goal of RL is to choose the policy which maximises the value of the accumulated reward that an agent achieves through its interactions with the environment. There are two broad ways of thinking about how to learn a policy using reinforcement. If we know something about how the environment works, then we can employ what is known as *model-based* learning, otherwise, we use *model-free* learning.

Model-free learning[5] is referred to as *tabula rasa* (clean slate) learning, since such methods approach the environment with no pre-existing knowledge, building their understanding entirely from successful and unsuccessful interactions with it. For this reason, model-free methods are the most versatile, and constitute the majority of recent research in deep reinforcement learning.

Two principal families of model-free learning are considered in this thesis: value-based and policy-based methods.

[5] As is the case for some model-based methods.

Value-based Methods

To understand value-based methods, we can consider Turing's chess algorithm. At each point, Turing calculates a "position value", comprised of the "material value" and the "position-play value". The next action is chosen as the one which leads to the highest position value. Although Turing's approach did work well for chess, it was not impenetrable, it was specific to chess and did not improve with additional experience. Instead, we will consider the value of a state as the expected, discounted value of the future reward.

For the policy π, we can formally define the state value, $V(s)$, at each time t in terms of the state s_t:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left\{ \quad \sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t, s_{t+1}) \quad \middle| \quad s_0 = s \quad \right\},$$

where $0 \leq \gamma < 1$ is known as the *discount factor*.

The value of γ is chosen to control how short-sighted the agent is. Values that are low represent a policy that cares only about immediate rewards, whereas values that are high represent policies that are far-sighted[6].

The value function tells us how desirable a state is, but we also need to determine the desirability of an action given a state. We do that by introducing the idea of *quality*. For any action $a \in \mathcal{A}$ at a state $s \in \mathcal{S}$, we can determine the quality of that action in that state:

$$Q(s, a) = \mathbb{E}_{\pi}\left\{ \quad \sum_{t=0}^{n} \gamma^t \mathcal{R}(s_t, a_t, s_{t+1}) \quad \middle| \quad a_0 = a, s_0 = s \quad \right\}. \tag{2.5}$$

Given a policy π, actions are chosen such that:

$$\pi(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s, a). \tag{2.6}$$

Applying the Markov property (Property 1) to Definition 2.5 yields the *Bellman recursion* [11]:

$$Q(s, a) = \mathcal{R}(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q(s', a'), \quad \text{where } s' = \mathcal{T}(s'|s, a). \tag{2.7}$$

From this, we can deduce the value of $Q(s, a)$ using an iterative approach: when an MDP in state s transitions to s' as a result of action a with reward r, determine the so-called *temporal-difference error* (or TD-error):

$$TD(s, a, r, s') = \left\{ r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q_i(s, a) \right\}. \tag{2.8}$$

[6]Values of $\gamma \approx 1$ can cause instability in infinite-horizon settings: changes in Q anywhere in the state space propagate globally, making convergence slow and sensitive to initialisation.

Then, we can update the estimate for $Q(s, a)$:

$$Q_{i+1}(s, a) = Q_i(s, a) + \alpha\, TD(s, a, r, s'), \tag{2.9}$$

where α is the *learning rate*, controlling the size of updates in stochastic systems.

This temporal-difference formulation directly parallels the biological reward prediction error encoded by phasic dopamine discussed in Section 2.2. Just as dopamine spikes when outcomes exceed expectations and dips when outcomes disappoint, TD error quantifies the discrepancy between predicted value $Q_i(s, a)$ and observed return $r+\gamma \max_{a'} Q(s', a')$. This neurobiological correspondence provides both inspiration and validation for TD-based learning algorithms: the same mechanism that evolution refined for adaptive behaviour in biological agents turns out to be a principled and effective basis for learning in artificial ones.

2.8.4 Deep Reinforcement Learning

The value-based framework developed above assumes tractable state spaces. Q-learning with tables stores $Q(s, a)$ for every state-action pair—feasible for gridworlds but not for real-world problems. Robotic control confronts continuous joint-space observations; visual tasks confront images with millions of possible configurations. Function approximation with neural networks becomes necessary. As discussed in the previous section, the central challenge is the *deadly triad* [10]: combining function approximation, bootstrapping, and off-policy learning creates training instability that naive gradient descent cannot handle.

Mnih et al. [12, 13] resolved this with Deep Q-Networks (DQN). By combining convolutional encoders with two stabilising techniques—*experience replay* (storing past transitions in a buffer and sampling them randomly to break temporal correlation) and *target networks* (a periodically frozen copy of the value network that provides stable TD targets)—DQN achieved human-level performance across 49 Atari games, learning directly from pixel observations with a single architecture and no game-specific engineering. This result established deep reinforcement learning as a practical methodology. DQN's architectural patterns—CNN encoders, replay buffers, target networks—underpin the methods used throughout this thesis. Its principal limitation is that the $\text{argmax}_a\, Q(s, a)$ operation requires discrete, enumerable actions; continuous control demands a different approach.

2.8.5 Policy Gradient Methods

Value-based methods derive policies implicitly via $\pi(s) = \text{argmax}_a\, Q(s, a)$, which requires enumerating all actions. This is tractable for discrete choices but intractable for continuous action spaces—robot joint torques, vehicle steering—where actions vary smoothly

over $\mathbb{R}^n$. Policy gradient methods address this by parameterising the policy directly as a distribution $\pi(a|s;\theta)$ and optimising θ to maximise expected cumulative reward. The *policy gradient theorem* [14, 15] shows that the gradient of expected return is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\, A_t\right], \tag{2.10}$$

where A_t is the *advantage*—how much better the chosen action was than the average for that state—estimated by a learned critic in actor-critic architectures [16]. This gradient can be computed without a model of environment dynamics, requiring only sampled trajectories.

Proximal Policy Optimisation (PPO) [17] is the de facto standard on-policy method. Its central challenge—that large gradient steps can catastrophically degrade policy performance—is addressed by clipping the probability ratio between the new and old policy to a narrow interval, naturally bounding the size of each update without expensive second-order constraints. PPO is simple to implement, stable across diverse tasks, and is the primary algorithm used in Chapters 5 and 7 of this thesis.

Soft Actor-Critic (SAC) [18] is the leading off-policy method. It augments the standard RL objective with a policy entropy term, rewarding diverse behaviour and improving exploration. Off-policy learning—reusing past experiences via a replay buffer—dramatically reduces the number of environment interactions required, an important property when sample collection is expensive or slow. SAC's combination of sample efficiency, entropy-regularised exploration, and stability makes it a suitable algorithm for continuous control when interaction cost is a constraint, as in some experimental settings in Chapter 8.

2.8.6 Sisyphus Plays Atari

Reward regimes which apply credit only upon completion of a specific task are known as *sparse*. A drone that receives reward only when it successfully navigates the forest must discover, through random interaction, the entire sequence of actions that leads there. Sparse rewards exacerbate the credit assignment problem [19]: it is difficult to determine which prior actions were responsible for a success that may have occurred hundreds of steps earlier. Classical examples include Montezuma's Revenge, an Atari game where meaningful rewards are obtained only after solving multi-step puzzles [12], and dexterous manipulation tasks where reward is withheld until grasp success [20]. In such settings, unguided exploration rarely yields the necessary action sequences, motivating the techniques discussed below.

Reward Shaping

When environment rewards are sparse or delayed, agents may require prohibitively many interactions before discovering rewarding trajectories. Reward shaping augments environment rewards with supplementary signals designed to guide learning towards desirable behaviours. A shaped reward function $\mathcal{R}'(s, a, s') = \mathcal{R}(s, a, s') + F(s, a, s')$ adds a shaping term F to the original environment reward $\mathcal{R}$.

Not all shaping functions preserve the optimal policy. Ad-hoc shaping—adding arbitrary bonuses for visiting certain states or taking particular actions—risks inducing policies that maximise shaped reward whilst failing to solve the original task. Potential-based shaping addresses this concern by constraining F to the form $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$ for some potential function $\Phi : \mathcal{S} \to \mathbb{R}$. This telescoping structure ensures that cumulative shaped reward across any trajectory equals the difference in potential between terminal and initial states, guaranteeing that optimal policies under shaped reward remain optimal under the original reward function.

The principle extends naturally to modular policy architectures. In distributed policy graphs, different units may receive divergent reward signals tailored to their specialised roles—one unit shaped towards "stabilise angle", another towards "minimise energy consumption". Careful reward design ensures units develop complementary skills rather than conflicting objectives. The potential-based guarantee provides a foundation for principled reward routing: shaping signals can guide unit specialisation during training without distorting the global objective that the complete graph must optimise.

Intrinsic Motivation

In their 2004 paper [21], Singh, Barto and Chentanez apply the idea of intrinsic and extrinsic motivation to reinforcement learning from psychology. As they describe it, "extrinsic motivation ... means being moved to do something because of some specific rewarding outcome, and intrinsic motivation ... refers to being moved to do something because it is inherently enjoyable". Developing reward metrics that rely on something other than environmental feedback makes training tractable in sparse reward settings.

Intrinsic motivation mechanisms encourage exploration by rewarding novelty, uncertainty reduction, or state visitation. Count-based methods reward visiting states inversely proportional to prior visits, encouraging agents to explore unfamiliar regions of state space. Curiosity-driven approaches reward prediction error: when an agent's internal model fails to predict the consequences of its actions, that unpredictability itself becomes rewarding, driving exploration towards surprising outcomes. Random Network Distillation provides a computationally efficient approximation: a fixed random network serves as a target, and a second network is trained to match its outputs. Prediction error is high for novel states the predictor has not yet encountered, making it a reliable proxy

for novelty.

These mechanisms address a fundamental tension in reward-driven learning. Environmental rewards reflect task objectives but may be too sparse to guide learning; intrinsic rewards are dense but may not align with task objectives. Combining extrinsic and intrinsic signals—weighting task reward against exploration bonuses—enables agents to learn efficiently in sparse domains whilst ultimately optimising for environmental objectives. The balance between these signals, and how that balance should evolve during training, remains an active research question with direct implications for distributed policy graphs: should individual units receive intrinsic motivation signals, and if so, how should unit-level exploration coordinate with graph-level task objectives?

Directed and Undirected Reward

Reward signals differ not only in sparsity but in their relationship to task objectives. Directed rewards explicitly encode goal achievement: reaching a target location, solving a puzzle, completing a manipulation task. These signals provide clear learning objectives but may induce specification gaming—agents that maximise reward through unintended means. Just as dopamine encodes incentive salience rather than genuine wellbeing, a reward function encodes the designer's *proxy* for success rather than success itself. Agents trained with poorly specified directed rewards may therefore develop pathological optimisation behaviours, maximising the reward signal through unintended strategies that fail to achieve the task designer's actual intent.

Undirected rewards, by contrast, encourage broad competence without specifying particular goals. Entropy maximisation rewards diverse behaviour; empowerment rewards states from which the agent can exert maximal influence over future states; skill discovery rewards the acquisition of distinguishable behavioural primitives. These approaches develop general capabilities that may transfer across tasks, but provide weaker guarantees about solving any specific objective.

The distinction matters for deployment. Directed rewards enable focused training on specific tasks but risk brittleness: agents may fail when task conditions differ from training. Undirected rewards develop flexible capabilities but may never achieve specific objectives reliably. Policy graphs offer a structural solution: directed rewards train specialised units for specific subtasks ("navigate to waypoint", "grasp object"), whilst graph-level coordination ensures these specialised capabilities compose into complete task solutions. The modular structure bounds specification gaming: even if an individual unit develops an unintended behaviour, explicit routing constraints limit how that behaviour propagates through the system.

2.8.7 Gödel, Escher, Bot: Emergent Behaviour and Bootstrapping

The heuristics introduced earlier in this chapter—bootstrapping, evolution, co-evolution—find concrete expression in reinforcement learning through mechanisms that generate training signal from the learning process itself rather than from external supervision. Self-play and curriculum learning exemplify this principle: agents improve by competing against past versions of themselves or by progressively tackling increasingly difficult tasks, creating virtuous cycles where capability improvements unlock access to new training experiences that drive further improvement.

Self-Play

When suitable opponents or training partners are unavailable, agents can learn by interacting with copies of themselves. Self-play embodies the co-evolutionary heuristic: as the agent improves, so does its opponent, maintaining appropriate challenge throughout training. The agent's current capabilities enable training experiences that develop new capabilities, which in turn unlock further improvement—a bootstrapping loop that can discover strategies beyond human conception, as demonstrated by AlphaGo's novel Go moves and OpenAI Five's Dota 2 team coordination.

However, self-play also risks pathological dynamics. Agents may develop strategies that exploit weaknesses in their current opponent (a past self) rather than developing generally robust capabilities. Cycling between brittle strategies—rock-paper-scissors dynamics—can prevent convergence to stable, high-quality policies. Maintaining population diversity and carefully managing the distribution of training opponents addresses these concerns, ensuring that self-play drives genuine capability improvement rather than narrow exploitation.

Curricula

Complex tasks may be intractable when attempted directly but learnable through careful sequencing of intermediate objectives. Curriculum learning presents tasks in order of increasing difficulty, allowing agents to develop foundational skills before confronting full task complexity. The principle mirrors human education: children learn arithmetic before calculus, basic motor skills before complex athletics.

Automatic curriculum generation extends this idea by adapting task difficulty to agent capabilities: rather than hand-designing task sequences, the curriculum itself becomes a learning problem. Approaches range from simple heuristics—training on tasks where the agent achieves intermediate success rates—to meta-learning methods that explicitly optimise curriculum parameters.

Domain randomisation represents a complementary approach: rather than sequencing tasks by difficulty, training exposes agents to diverse variations of the same task. Randomising physics parameters, visual appearances, or environmental configurations encourages policies that generalise across conditions rather than overfitting to specific settings.

Chapter 6 extends these ideas through procedural environment generation. Rather than manually designing curriculum stages or randomisation distributions, the EnvCraft system generates thousands of validated training environments from natural-language specifications. This enables systematic study of how training diversity affects generalisation—a question central to deploying reinforcement learning beyond narrow benchmark distributions.

2.8.8 Hierarchical RL

The foundational concepts developed earlier in this chapter—division of labour from pin factories, reward prediction error from dopamine neuroscience, System 1/System 2 cognitive dichotomies—all point towards a common architectural principle: complex adaptive systems benefit from modular decomposition along functional boundaries. Reinforcement learning research has explored this principle through hierarchical frameworks that decompose policies into reusable, composable units.

Options [22] extend the action space to include temporally extended actions—policies that execute over multiple timesteps until a termination condition is met. An option consists of three components: an initiation set (states where the option can start), a policy (what actions to take), and a termination condition (when to return control). This formalism enables agents to learn at multiple temporal scales: low-level options learn motor skills ("grasp object", "move to location"), whilst high-level policies learn to compose these skills into task solutions. Options align with the chunking mechanisms discussed in Section 2.3: just as practiced skills become automatic routines, learned options become reusable behavioural primitives.

Feudal RL [23, 24] introduced explicit hierarchical control through manager-worker relationships. Managers operate at slower timescales, setting subgoals and decomposing tasks; workers execute low-level policies to achieve these subgoals. This mirrors the division of labour in pin factories: managers coordinate specialisation, workers execute specific skills. Feudal RL demonstrates that hierarchical value function decomposition—where managers learn to evaluate subgoal achievement and workers learn skill execution—can improve learning efficiency in complex domains.

MAXQ [25] formalises hierarchical decomposition through recursive task decomposition. A task graph defines subtasks and their constraints; each node in the graph has a value function decomposed into completion value (reward for completing this subtask)

and continuation value (reward from parent tasks after completion). This decomposition enables state abstraction: subtask policies need only observe state features relevant to their specific objective, reducing the effective state space. MAXQ exemplifies "carving nature at the joints" (Section 2.1): task decomposition should align with natural problem structure rather than arbitrary boundaries.

HAM (Hierarchical Abstract Machines) [26] represents hierarchical policies as partially-specified finite state machines. Each machine defines legal action sequences through states, transitions, and choice points where learning occurs. Non-choice states execute deterministically; choice states invoke learned policies or sub-machines. HAM provides stronger constraints than options or MAXQ: the hierarchy itself encodes domain knowledge about valid action sequences, reducing the space of behaviours the agent must explore. This connects to Torres's conception of automation (Section 2.4): constraints imposed at construction time enable efficient operation within bounded domains.

Despite their conceptual elegance, these hierarchical frameworks face deployment challenges that limit real-world applicability:

1. **Co-location assumption**: Prior work assumes hierarchy components share a process and memory space. Options, feudal managers, MAXQ subtasks, and HAM machines all presume instantaneous communication and shared state access. This precludes physical distribution across heterogeneous hardware—exactly what System 1/System 2 dichotomies suggest (reactive edge control, deliberative cloud reasoning).
2. **No network/communication model**: Existing frameworks lack explicit models of inter-component communication. Delegation, return, and reward routing occur implicitly through shared memory. Real deployment confronts latency, jitter, packet loss, and partial failures—conditions these frameworks do not address.
3. **Limited interpretability and accountability**: Soft attention mechanisms and implicit routing make it difficult to trace which component made which decision. When a policy fails, identifying the responsible unit requires analysis of learned attention weights rather than explicit call traces. This undermines the debugging and auditing requirements for safety-critical deployment.
4. **Training complexity**: End-to-end training of hierarchical policies requires differentiating through routing mechanisms and managing credit assignment across temporal abstractions. This couples learning across components, preventing independent development and deployment of specialised units.

Policy graphs, introduced in Chapter 5, extend hierarchical RL to address these four deployment gaps. A directed graph $G = (V, E)$ of callable policy units uses hard routing and call-and-return semantics: exactly one unit is active at any moment, commitment

bounds ($k_{\min}, k_{\max}$) prevent unstable switching, and explicit call traces provide accountability that soft-attention hierarchies cannot. Units execute as physically distributed networked services—reactive control on low-power edge devices, deliberative reasoning on remote hardware—operationalising the System 1/System 2 distribution described in Section 2.3. Chapter 8 extends this further through CALF, treating network conditions as first-class training objectives so that policies learn to tolerate the latency and packet loss they will encounter at deployment.

2.9 Systems II

Having established the algorithmic foundations of reinforcement learning, we turn to the distributed systems context in which policy graphs must operate.

Remote Procedure Call (RPC) systems enable function invocation across network boundaries, presenting remote execution with the appearance of local function calls. Traditional RPC assumes reliable, low-latency networks—this holds within data centres but fails across Wi-Fi, cellular, and satellite links, where variable latency, jitter, packet loss, and partial failure are normal. Distributed systems must anticipate component failures and asynchronous delivery. Fault domains define boundaries within which failures correlate; placing time-critical policy units locally and computationally intensive deliberation remotely ensures that network failures degrade gracefully rather than causing total system collapse.

Observability and traceability become critical when distributed systems fail. When a deployed policy fails, operators must identify which unit made which decision, under what observations, and whether the cause was learned behaviour, network failure, or hardware fault. Policy graphs' hard routing and call-and-return semantics provide this traceability: execution traces explicitly record which units were active and when delegations occurred. This accountability distinguishes policy graphs from soft-attention hierarchies where responsibility diffuses across learned weights and cannot be inspected.

Containerisation packages code and runtime dependencies into portable units that execute consistently across diverse hardware. This enables deployment parity: the same policy code executes in pure simulation, simulation with network models, and real hardware, eliminating discrepancies between training and production environments. Chapter 8's CALF framework leverages containers for exactly this purpose.

Where pin factories achieved productivity through specialisation—eighteen workers performing distinct operations—policy graphs achieve deployability through modular accountability: eighteen policy units performing distinct behaviours, each traceable, testable, and independently deployable. The architectural patterns from engineered systems reviewed in Chapter 3—A320 flight computers distributing responsibility across ELACs and SECs, power grids coordinating IEDs at substations with SCADA at na-

tional scale—inform this design: reliability emerges from constrained transitions between well-defined components, not from monolithic optimisation of opaque end-to-end systems.

This chapter has assembled the conceptual vocabulary on which the remainder of the thesis depends. Beginning with Adam Smith's observation that dividing labour along natural joints dramatically multiplies productive capacity, and following the thread through Skinner's reward schedules, dopamine's encoding of prediction error, Kahneman's dual-process cognition, Torres's *discernimiento*, and Turing's notion of positional value, we have arrived at an account of why reinforcement learning is structured as it is and what it still lacks for real deployment. The frameworks of Options, Feudal RL, MAXQ and HAM are elegant, but they were designed for co-located, monolithic execution; the distributed, heterogeneous, failure-prone deployment environments of the real world demand something more explicit. The following chapters address that gap: the Lessons chapter grounds these abstractions in three engineering systems where distribution and failure have proved consequential; the Works chapter surveys the state of real-world RL deployment; and the research chapters introduce policy graphs, compact edge-deployable encoders, communication-aware training, and procedural environment generation as components of a practical answer to the question Turing could not yet resolve.

Chapter 3

Lessons

Time delay has long been recognised as a defining systems constraint rather than a minor implementation detail—Sheridan's 1993 review of space teleoperation made exactly this argument, and this chapter takes a similarly broad view: before turning to case studies, it sketches the recurring design patterns—redundancy, sensor fusion, failsafes, distributed control, protections, and latency—that allow automated systems to operate safely outside the laboratory.

3.1 Redundancy

Redundancy is a fundamental design principle in real-world systems: it involves including additional components, subsystems, or resources beyond those strictly necessary to perform a task. These additions can either remain inactive until a primary component fails—cold redundancy—or operate alongside the primary so that handover is immediate—hot redundancy. Three distinct forms arise in practice: *structural redundancy* duplicates physical components (motors, processors, sensors); *functional redundancy* provides different subsystems capable of fulfilling the same function via varied mechanisms; and *informational redundancy* adds extra data or signals to support error detection and cross-checking.

Redundancy is most effective when component failures are statistically independent. In practice, however, many systems face correlated failure modes that undermine this assumption. Redundant transformers may fail simultaneously during a heat event; redundant sensors may all be affected by the same electromagnetic interference; processors from the same manufacturing batch may degrade at the same rate under the same thermal stress. Diversity—using components from different vendors, routing communication over physically separate paths, or implementing diverse software architectures—reduces the likelihood of simultaneous failure and maintains the intended protective value of redundancy.

Figure 3.1: Examples of redundancy in real-world systems. The figure contrasts structural, functional, and informational redundancy across several domains, showing how backup components, overlapping roles, and cross-checkable signals preserve operation when individual elements fail.

Sensor redundancy is frequently combined with majority voting: if three sensors measure the same physical variable and one produces a deviant reading, the system discards the anomalous input and continues using the remaining two. Figure 3.1 illustrates eight real-world implementations. For reinforcement learning deployed in real-world settings, redundancy corresponds to the ability to maintain a functioning policy under partial component failure—a principle directly embodied in the distributed execution model of Chapter 5.

3.2 Sensor Fusion

Sensor fusion combines data from multiple sensors to produce a unified and more reliable representation of the environment. It is essential in systems that must operate under uncertainty, noise, or partial observability. Several well-established algorithms underpin practical implementations: Kalman filters and their nonlinear variants (EKF, UKF) provide mathematically grounded methods for continuous state estimation; Bayesian approaches offer a broader probabilistic framework; and particle filters or data-driven models are used where precise modelling is difficult or computational efficiency is paramount.

Autonomous vehicles illustrate the complementarity that motivates sensor fusion: cameras capture rich visual information for object classification; LiDAR provides high-resolution 3D spatial data; radar performs reliably in adverse weather; and GPS supports global localisation. Manufacturers such as Waymo integrate all four modalities into their core architecture, achieving more consistent and resilient performance than any single sensor could provide. In robotics, Boston Dynamics platforms fuse IMU data with stereo vision and force feedback to achieve dynamic locomotion over uneven terrain. Figure 3.2 illustrates how these input modalities combine across application domains.

Sensor fusion can be centralised—raw data transmitted to a single processing unit for joint optimisation—or decentralised, with local nodes performing preliminary fusion before passing results to a coordinator. Centralised architectures allow tighter integration but demand significant bandwidth; decentralised designs reduce communication overhead and support fault tolerance at the cost of synchronisation complexity. For reinforcement learning, sensor fusion corresponds to partial observability management: the observation function $\mathcal{O}$ aggregates heterogeneous inputs before the policy acts, and the architecture of that aggregation affects both accuracy and latency.

3.3 Failsafe

A failsafe is a design feature intended to move a system into a safe state when a fault is detected. Unlike redundancy, which attempts to maintain normal operation despite

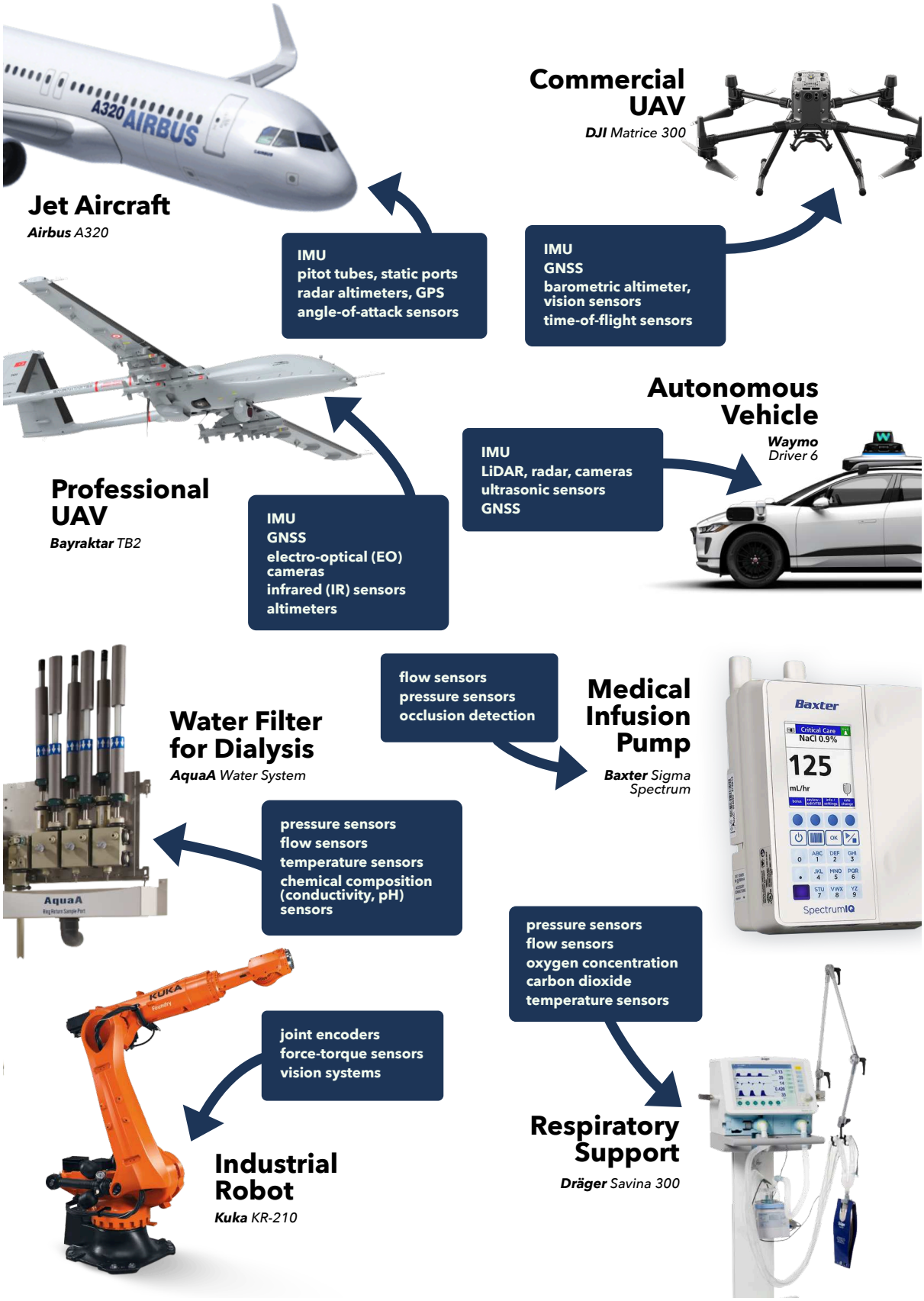


Figure 3.2: Some examples of real-world systems and the inputs that they use with sensor fusion. Many applications use an Inertial Measurement Unit (IMU), a sensor system that uses accelerometers, gyroscopes, and sometimes magnetometers to measure linear acceleration, angular rate, and orientation.

component failures, a failsafe prioritises safety over continued function. Two archetypal responses illustrate the range: a robotic arm that halts immediately when it detects unexpected resistance, and an aircraft that transfers control to the pilot when automation encounters a fault it cannot resolve. In both cases, the system acts to bound harm rather than to preserve performance.

Failsafes are activated by *cross-checks*: multiple independent sensors or monitors are consulted and compared. If two navigation systems report conflicting positions, the autopilot may disengage and alert the flight crew. Triggers may also be timeouts (a process fails to complete within an expected window), out-of-range readings, or loss of communication. When one of three sensors produces a deviant reading, a majority voting scheme can continue operation using the remaining two; when disagreement is severe or irresolvable, the system transitions to the safe state. In nuclear power plants, that state is unambiguous: control rods drop into the reactor core automatically, shutting down the reaction without waiting for human intervention. The "safe state" is always defined relative to the application's risk profile—shutdown for some systems, a conservative limited-function mode for others.

Failsafes are implemented in hardware, software, or both. Hardware failsafes—pressure-relief valves, mechanical interlocks—address physical failure modes independently of software. Software failsafes respond to a broader range of conditions but are themselves susceptible to bugs; "watchdog" hardware addresses this by monitoring software execution and enforcing a reset if anomalies are detected. For reinforcement learning, failsafe semantics correspond to safety-constrained terminal states: the policy must be designed to reach or respect them, not to optimise through them.

3.4 Distributed Systems

As Adam Smith's pin factory illustrated, modern systems achieve efficiency through distributed specialisation: components are produced or processed separately and brought together only at the point of integration. Within a single autonomous vehicle, multiple subsystems—perception, navigation, control, communication—operate in parallel on separate hardware units, each optimised for its own task. Coordination across these units demands robust protocols for synchronisation and fault management, but the separation itself is what makes specialisation possible.

Hierarchy is the organisational principle that makes large-scale distribution manageable. In industrial process control, low-level programmable logic controllers (PLCs) manage rapid, time-sensitive operations—valve control, temperature regulation—whilst supervisory control and data acquisition (SCADA) systems handle high-level monitoring and decision-making across entire facilities. Air traffic management extends this to three levels: local aircraft systems manage immediate flight dynamics; pilots handle tactical

situational awareness; centralised ATC coordinates regional and national airspace. Each level operates quasi-independently, which means failures at one level are isolated and do not cascade automatically into the others.

Distribution enhances transparency and resilience because defective subsystems can be examined in relative isolation. Spreading functionality across multiple independent units makes faults easier to diagnose, facilitates targeted maintenance, and supports natural redundancy through overlapping roles. For reinforcement learning in real-world settings, this hierarchy of timescales and responsibilities directly motivates the modular policy architecture examined in later chapters. The case studies that follow show how these general principles are realised under much tighter operational and safety constraints.

3.5 Protections

Protections are mechanisms that enforce operational boundaries and prevent unsafe actions. They constrain system behaviour to remain within safe or allowable parameters, acting as safeguards against misuse, malfunction, or unforeseen environmental influences.

In aviation, *flight envelope protections* are the defining example. Fly-by-wire aircraft include control laws that prevent the aircraft from exceeding critical aerodynamic or structural limits—excessive pitch, bank angle, airspeed, angle of attack, or load factor. In an Airbus aircraft operating under normal law, the system actively limits control inputs that could induce a stall or over-G condition. Such protections reduce pilot workload by embedding aerodynamic constraints directly into the control interface; the pilot works within a pre-constrained envelope rather than manually avoiding its boundaries.

Energy systems rely on analogous layered schemes. In electrical grids, protective relays monitor voltage, current, and frequency to detect abnormal conditions such as short circuits or overloads; when triggered, they isolate the affected section to prevent wider instability. In nuclear power plants, automatic shutdown sequences engage if critical parameters exceed safe limits, ensuring the plant enters a safe state without requiring human intervention. For reinforcement learning, protections correspond to constrained action spaces and safety-critical terminal states—the policy graph architecture in Chapter 5 enforces analogous boundaries through hard routing and commitment bounds. The following section turns from the spatial constraints imposed by protections to the temporal constraints imposed by latency.

3.6 Latency

Latency is the time delay between an input or event and the corresponding system response. In remote operations—UAVs, telesurgical instruments, bomb disposal robots—delays of even a few hundred milliseconds can result in misalignment, errors, or loss of

situational awareness, because operators rely on immediate visual, haptic, or auditory feedback to guide precision tasks. Low latency is often framed as the primary goal, but network congestion introduces jitter: variability in delay that disrupts the timing of control loops and is often more detrimental to performance than moderate, stable delay [27].

Recent evidence makes this concrete. Noguera Cundar (2023) [28] examined latency variability in a teleoperated ultrasound robot: under standard WLAN, latency fluctuated between 33 ms and 750 ms, producing a maximum positional error of 7.8 mm. Switching to an isolated VLAN reduced the average delay by approximately 200 ms and cut positional error by 70%, to 2.4 mm. Kelkkanen (2023) [29] found a complementary result in a VR aiming task: unpredictable target motion impaired performance at around 90 ms latency, whilst predictable motion extended the usable threshold to approximately 130 ms. Together, these studies demonstrate that consistent, predictable latency is more important for precise real-time control than achieving the lowest possible but highly variable delay.

Systems with critical latency requirements manage this through two complementary strategies. Dedicated links—leased lines or isolated VLANs—provide fixed bandwidth and temporal determinism at the cost of infrastructure overhead [30]. Local fallback mechanisms provide the alternative: equipping the local device with onboard autonomy to hold position, return to a safe waypoint, or follow a preprogrammed routine when real-time remote input is unavailable [31, 32]. Systems establish clearly defined latency thresholds—derived from empirical testing or risk analysis—beyond which the device transitions to a fallback mode rather than continuing degraded remote operation.

Effective latency management therefore depends not only on minimising delay but on maintaining stability and predictability under diverse conditions. Well-managed latency allows users to form accurate mental models of system behaviour, enhancing both trust and task performance [33, 34]. Distributed control and robust fallback mechanisms are the principal engineering responses, and both recur in the case studies that follow.

3.7 Case Study: Airbus A320

The Airbus A320 marked a turning point in civil aviation by introducing the first fully digital fly-by-wire (FBW) control system in a commercial airliner. Rather than mechanically linking the cockpit controls to the flight surfaces, the FBW architecture interposes digital flight control computers: pilot inputs are interpreted, filtered, and translated into surface commands according to the active flight law. This computer-mediated control reduces pilot workload, enables envelope protections, and supports structured degradation when faults occur. The A320 offers an unusually well-documented case study in how layered automation, distributed computation, and redundancy are engineered into a

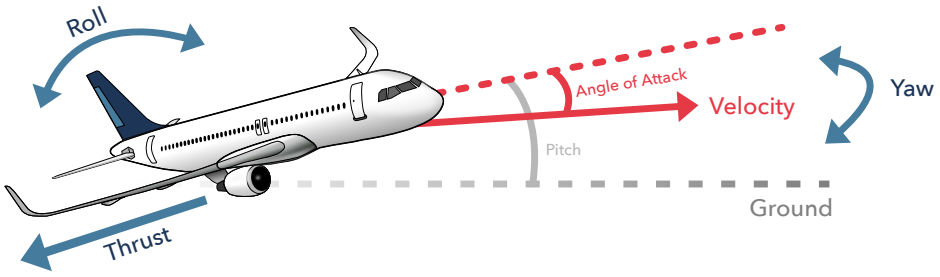


Figure 3.3: The A320's principal flight axes: pitch (nose up or down), roll (wing tilt), yaw (nose left or right), and the angle of attack (AoA). AoA is continuously monitored by the flight envelope protection system and limited under normal law to prevent aerodynamic stall.

system that must remain safe across a wide range of failure conditions.

The A320 replaced traditional control columns with side-sticks, which transmit pilot inputs to the FBW system. The FBW system determines the desired aircraft state and issues appropriate commands to the flight control surfaces; rudder pedals similarly feed through digital flight control computers rather than direct mechanical linkage.

The A320 flight deck relies on a clear separation of roles between the flight crew and the onboard systems. Typically, two pilots operate the aircraft: the *pilot flying*, responsible for flight path control, and the *pilot monitoring*, who manages communication, systems monitoring, and checklist execution. This human division of labour is mirrored in the architecture of the computerised part of the A320, which distributes responsibilities across multiple functionally distinct computers. A broad overview of the different systems in the A320 is shown in Figure 3.4, along with the different forms of communication that exist between them. Elevator and Aileron Computers (ELACs) manage the aircraft's primary pitch and roll control, whilst Spoiler and Elevator Computers (SECs) serve as a backup for pitch and roll and control the spoilers. The Flight Control and Guidance Computers (FCGCs or FCCs) process autopilot, flight director, and auto-thrust commands. These separate systems each handle discrete but interdependent functions, ensuring that no single failure compromises overall control. The structured separation of roles—both human and computational—enhances system transparency, reduces operational workload, and improves the ability to isolate and manage failures.

Structural redundancy is critical to the safe operation of the A320. The aircraft does not depend on any single point of control or measurement. Instead, it uses multiple, independent pathways to perform critical functions. This principle extends across both hardware and information domains. The aircraft incorporates redundant sensors—such as multiple angle of attack vanes, pitot tubes, and inertial reference systems—which feed

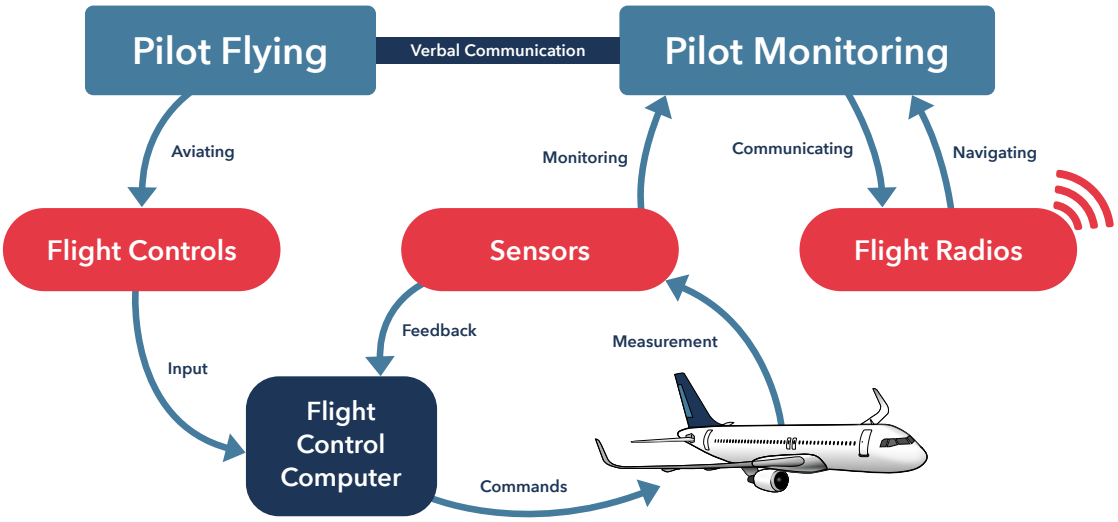


Figure 3.4: Overview of the A320's principal operational relationships. The cockpit includes at least two members of flight crew: the captain and the first officer, dividing duties between the pilot flying and the pilot monitoring. The pilot flying sends commands to the flight control computer (FCC), which interprets them according to the active flight law (Figure 3.7) and acts in a feedback loop with the A320's sensors.

data into the flight control computers. The system compares these inputs to identify inconsistencies, known as 'cross-checking'. When the data from one sensor differs significantly from the others, the system flags the anomaly to the operator and excludes the faulty input from critical calculations. This form of informational redundancy maintains system integrity even amidst degraded or failed components.

Flight control computers in the A320 also implement structural redundancy. The ELACs and SECs each operate in pairs and each FCC includes multiple processing lanes. If one processor or module fails, another can assume its function without disrupting flight or causing safety issues. The system performs a cross-check, monitoring the consistency of outputs across computers in real time. When a disagreement arises between redundant units, internal logic can determine the correct value, such as through voting or by excluding outliers. This approach allows the aircraft to isolate and contain errors before they affect downstream systems. These redundancies do not just act as backups; they form a distributed system for efficient fault detection, identification, and mitigation. The flight control system is illustrated in Figure 3.5.

Hydraulic systems, responsible for powering the flight control surfaces, landing gear, brakes, and thrust reversers also follow a similar redundancy model. The A320 includes three fully independent hydraulic circuits, colour-coded green, yellow, and blue. Each system draws power from different sources. Figure 3.6 shows how the source from which each system is initially derived and from which each can derive backup power. The

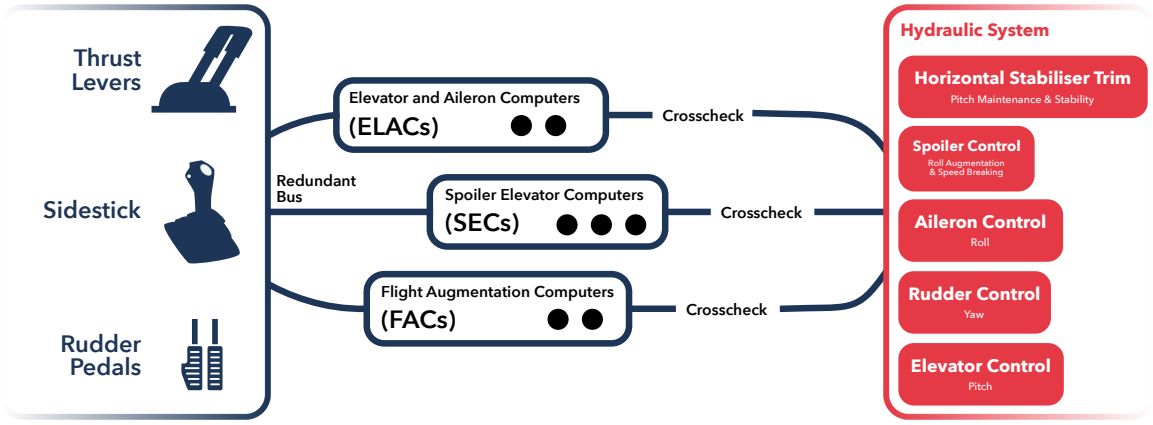


Figure 3.5: The Airbus A320 is controlled by the pilot flying. It implements redundancy through the use of multiple buses from the flight controls to the various flight computers. Each flight computer runs in a redundant set. The spoiler elevator computer (SEC) runs in triplicate, whereas the elevator and aileron computer (ELAC) and flight augmentation computers (FAC) run in duplicate. Usually, each of the systems will have redundant computers manufactured by different facilities, in order to reduce the risk of a correlated error affecting all of the units. If a cross-check fails, the system may move the A320 into 'alternate mode' and will alert the pilots. The hydraulic system implements the decisions made by the flight computers and is, itself, protected by redundancy as seen in Figure 3.6.

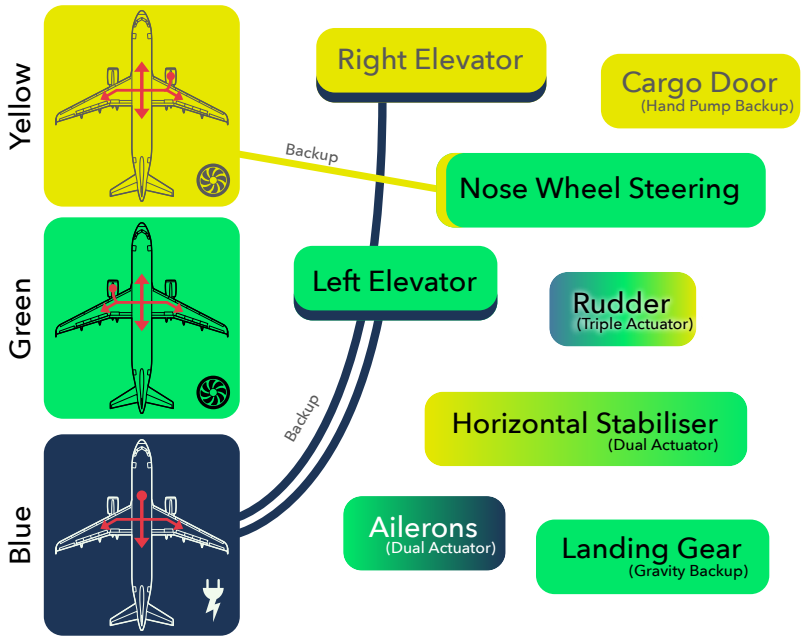


Figure 3.6: The A320's three independent hydraulic circuits—green, yellow, and blue—each with separate power sources. Green and yellow are engine-driven; blue is electrically powered, with a Ram Air Turbine as emergency backup. Gradients indicate dual-actuator (hot-redundant) surfaces; the Power Transfer Unit provides backup for the elevators. The rudder has triple actuators.

green and yellow systems rely on engine-driven pumps, whilst the blue system is powered by an electric pump. In some scenarios, such as electrical failure, the blue system can also receive power from a Ram Air Turbine (RAT), which deploys into the airstream to generate hydraulic pressure. These three systems supply overlapping sets of actuators, ensuring that each critical control surface remains operable even if one or two hydraulic systems fail. The separation of hydraulic power sources prevents localised mechanical failure from propagating into system-wide control loss.

Engine redundancy is another important feature of modern aircraft. Although designed for optimal performance with both engines functioning, the A320 can safely complete a flight with only one operational engine. The remaining engine provides sufficient thrust for level flight and controlled descent, whilst key systems remain powered through electrical and hydraulic cross-connections. The FBW architecture adapts automatically to the new flight envelope, adjusting control laws to ensure safe handling. The functional redundancy of having two pilots also means that, if one pilot becomes incapacitated, the aircraft can continue to be flown safely. In this way, the A320 maintains functional integrity through layered redundancy, enabling safe and managed degradation rather than catastrophic failure.

In addition to hardware and sensor redundancy, the A320 employs a layered hierarchy of flight control modes known as flight laws. These laws—normal, alternate, direct, and mechanical backup—govern the behaviour of the fly-by-wire system under different operational conditions. Figure 3.7 summarises each mode and the conditions under which transitions between them occur. In normal law, the system enforces full flight envelope protections: pitch, load factor, bank angle, and stall prevention. If certain failures occur, the system reverts to alternate law, which preserves some protections whilst allowing wider pilot authority. Direct law removes all protections, translating side-stick inputs directly to control surface movements (Figure 3.8). In rare cases of complete electronic failure, mechanical backup allows direct control of the flight surfaces. This tiered approach ensures that pilot authority increases progressively as system assistance decreases, allowing continued control in a wide range of degraded scenarios.

Pilots operating the A320 are trained to prioritise *aviate, navigate, communicate*: safe flight first, then navigation, then communication. This priority ordering is embedded in procedure, not just advice, and reflects the same principle as the flight laws themselves: the most safety-critical function is preserved at every level of system degradation.

The Airbus A320 illustrates how layered architecture, distributed computation, and structured redundancy can be engineered into a system that manages complexity without sacrificing transparency or safety. Crucially, its design anticipates off-nominal conditions, incorporating mechanisms to absorb and adapt to them without loss of safe operation. Several principles emerge with broad relevance to automated systems. The separation of roles—between pilots, between software components, and across hardware layers—

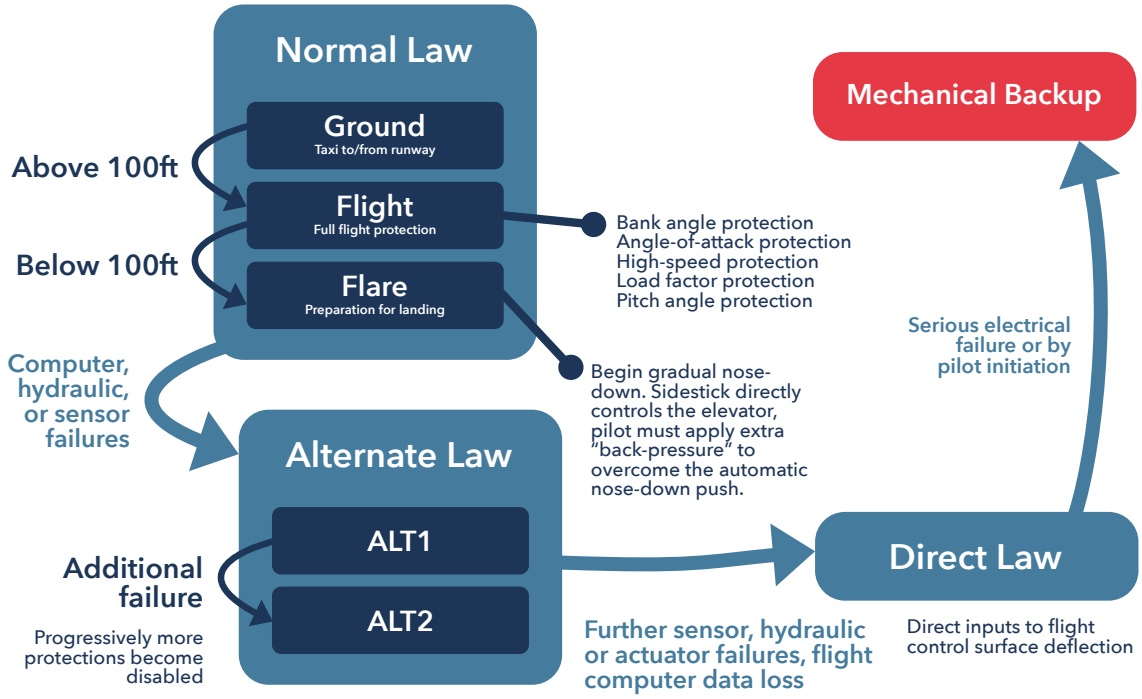


Figure 3.7: The Airbus A320 uses four flight-control modes: *normal law*, *alternate law*, *direct law*, and *mechanical backup*. Most flights remain in normal law throughout, where many protections are used to make aviation easier for pilots and ensure safety. If the onboard system detects an issue, such as a failed sensor cross-check, alternate law will engage. In alternate law, many protections still exist, but certain limits are removed. If the issues continue or worsen, the system moves to direct law, in which, among other changes, the pilot's side-stick directly commands the aircraft actuators to bypass potentially erroneous onboard calculations. If the system experiences a serious failure, the pilot can directly control certain mechanical features of the A320, bypassing electrical systems entirely.

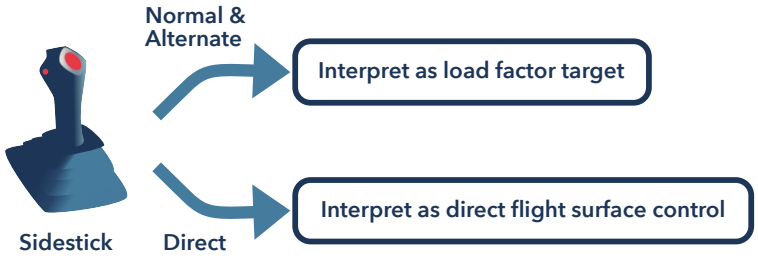


Figure 3.8: The side-stick in the Airbus A320 allows pilots to control pitch and roll. In normal and alternate laws, the amount the pilot pulls back or pushes forward is translated into a target load factor. This means that a modest pull produces a slight increase in load factor (and thus a modest pitch-up effect), whilst a larger pull commands a higher load factor up to the envelope limits. Lateral movements of the stick control roll—moving the stick left initiates a left bank, and moving it right initiates a right bank. Under direct law, the stick input bypasses intermediate global computations, since the flight envelope protections are no longer required and since the FCCs and sensors may not be sufficiently trusted or functional to reliably perform load factor calculations. Instead of being interpreted as a target load factor, the side-stick's position is directly sent to the control surface actuators.

supports clarity of function and traceability in the event of anomalies. Redundancy, both informational and structural, ensures that no single failure in a sensor, computer, or control surface propagates into wider system failure. The tiered flight laws define constrained transitions between operational modes: each law specifies what protections are active, what the pilot controls mean, and under what conditions the system degrades to the next level. This is the direct structural precedent for the constrained transition semantics of policy graphs in Chapter 5.

The Kangduo surgical robot presents a different realisation of these principles, where the central challenge is not flight envelope protection but latency management across a network.

3.8 Case Study: Kangduo Surgical Robot

Operating increases a surgeon's hand tremor by approximately 8.4 times compared to desk work [35, 36], and traditional open surgery forces prolonged awkward postures that increase musculoskeletal strain [37]. Patient travel for medical treatment costs Americans $89bn per year [38], much of which reflects the need for specialist procedures to be concentrated at major centres. Robotic surgical systems—standardised by the da Vinci platform since the early 2000s—address these constraints by providing tremor filtration, motion scaling, and ergonomic consoles [39]. However, a typical da Vinci system costs

$1.5–2 million, limiting adoption to well-resourced institutions. The Kangduo KD-SR-01, first registered by China's National Medical Products Administration in 2022, was designed as a substantially lower-cost alternative for the domestic Chinese healthcare market. It incorporates the core elements of master–slave robotic surgery—three-arm configuration, ergonomic open console, and high-definition 3D imaging—and is instructive for this thesis because of what its telesurgical deployments reveal about latency.

Clinical evaluations have reported acceptable safety and efficacy across prostatectomy, nephrectomy, and urological repair [40, 41, 42]. The evidence supports cautious claims about feasibility and surgeon ergonomics in the reported settings; the focus here is on the networking constraints that emerge when the system is extended to telesurgical use.

Published descriptions of the KD-SR-01 indicate that it combines tremor filtration with motion scaling to improve fine control. The available literature does not provide enough low-level implementation detail to justify a stronger claim about the precise filtering method, but it is clear that the system is designed to smooth surgeon input and support delicate manipulation.

The KD-SR-01 system is designed with an integrated dual-control interface that allows operative control to pass from the primary surgeon to a secondary surgeon. Figure 3.9 shows the relationship between different ways in which the system can be controlled. In this configuration, the primary surgeon initially manipulates the robotic instruments from the master console; during critical phases of the procedure, the system permits the secondary surgeon to assume control. This handover is presented as a synchronised transition intended to avoid disrupting the flow of the operation.

Robotic subsystems—including arms, actuators, and endoscopic tools—must continuously perform self-diagnostics to detect anomalies, initiate fail-safe modes, or prompt handover to manual control where necessary. Handover mechanisms themselves need to be secure and non-disruptive, allowing rapid transition of control authority without interrupting the surgical workflow.

The published material is more informative about operational features than about low-level implementation. The safest conclusions are therefore limited ones: tremor filtering, motion scaling, workspace constraints, and rapid handover are treated as safety-relevant features, but the available papers do not provide enough detail to support stronger claims about the internal power, filtering, or fieldbus design of the platform.

The KD-SR-01 system also depends on robust networking to facilitate real-time communication between its master console and the robotic arms. The reported studies use dedicated wired links or mixed 5G and wired configurations, which is enough to show reliance on stable low-latency communication, but not enough to specify the underlying real-time networking stack in stronger terms.

Latency management is extremely important. A recent clinical and animal study by Fan et al. (2023) [43] evaluated the feasibility of dual-console telesurgery using both

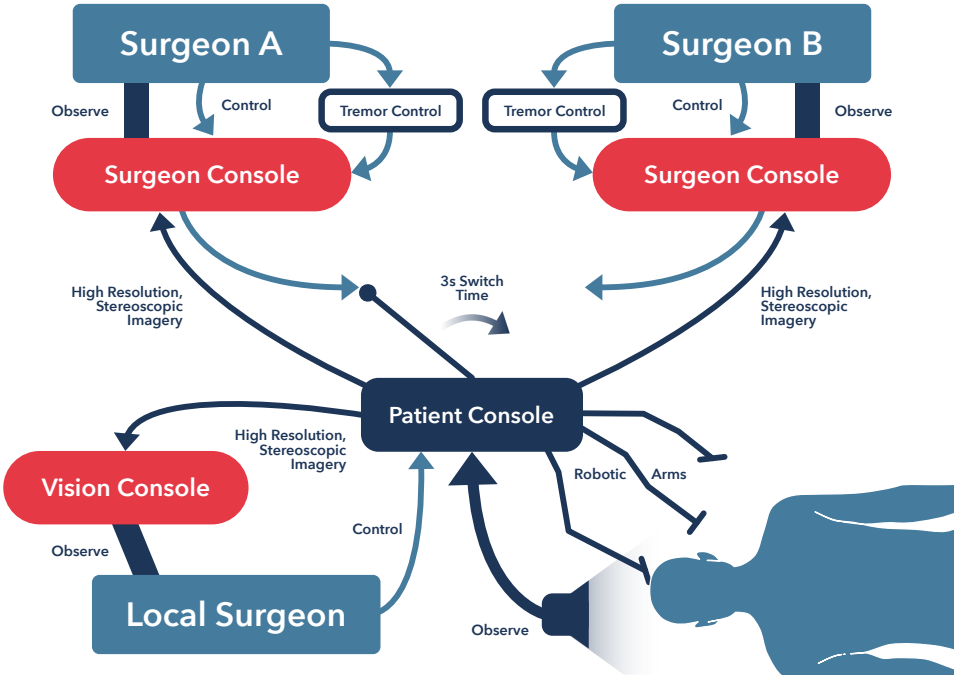


Figure 3.9: The Kangduo surgical robot can be controlled by several different operators. If two remote surgeons are involved in an operation, one may observe whilst another controls the robotic arms from the patient console. In longer operations, or during training, one surgeon may take over from another; in the Kangduo, this handover occurs in less than three seconds. As with the Airbus A320, the surgeons' actions are not typically passed directly to the robot actuators, but instead undergo smoothing for tremor removal and other safety checks. A feedback cycle then links the patient console, the robotic arms, and the observation stream from cameras and sensors. A local surgeon can also intervene directly via the patient console using in-person observation and the stereoscopic view available on the local vision console.

wired and fifth-generation (5G) networks. The experimental networks for the wired and 5G trials are shown in Figure 3.10. In the animal model, partial nephrectomy was performed remotely over an 80 km distance via a dedicated Internet Leased Line, achieving a mean latency of 130 ms (range 60–200 ms) and a control swap time of just 3 seconds, with no observed complications or packet loss exceeding 1%. In a subsequent human trial, a 32-year-old patient underwent remote pyeloplasty with a mixed 5G and wired network configuration. The mean latency reached 271 ms (range 206–307 ms), but performance remained within clinically acceptable limits in that setting. The study treated latency below 300 ms as workable for telesurgery, whilst also noting that lower values are preferable for longer procedures or critical surgical steps.

One of the primary technical challenges in telesurgical systems such as KD-SR-01 lies in ensuring stable, predictable latency. While average latency below 300 ms may be clinically acceptable, variability—known as jitter—can severely impact surgical precision and operator confidence. Even when nominal delay falls within thresholds, sudden spikes can desynchronise hand–eye coordination or disrupt visual feedback, particularly in procedures requiring fine motor control. As demonstrated in the dual-console clinical trial, latency values ranged from 206 ms to 307 ms, with a mean of 271 ms.

The KD-SR-01's dual-console design introduces an important layer of redundancy: should the remote surgeon experience network failure or degraded responsiveness, local control can resume within three seconds. This sub-three-second swap is relevant not only to training and oversight but to continuity of care under degraded communication conditions.

The published reports also make clear that latency is treated as an operational safety variable, even if they do not fully document the console-level monitoring interface. That is enough for the present argument: stable communication is part of the control problem rather than a background implementation detail.

Telesurgical systems like the Kangduo provide a useful example of automation operating within a real clinical service. The Kangduo acts as one node in a wider operational system involving a human surgeon and several other clinical systems in the operating theatre. The roles are clearly defined: the primary surgeon makes decisions and acts, the network communicates those decisions, the Kangduo interprets the command, applies tremor smoothing and safety checks, and sends high-quality 3D imaging back to the operator.

Experimental telesurgical deployments of the system have shown that whilst low latency remains desirable, consistent and stable latency is more important for operational safety and human performance. The dual console design and presence of a local fallback provide important redundancy. The required presence of a surgeon with the patient means that, even in the case of a network failure or a serious mechanical issue, a qualified human operator can intervene and prevent complications arising.

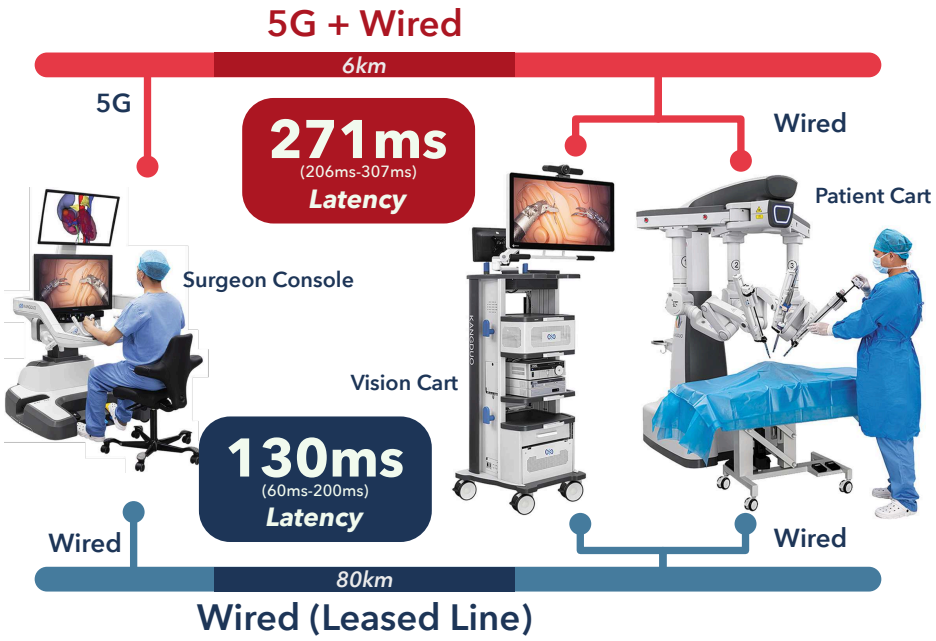


Figure 3.10: Experimental setups for dual-console telesurgery using the Kangduo KD-SR-01, as described by Fan et al. (2023) [43]. The wired configuration (blue) used an internet leased line over 80 km, achieving 130 ms mean latency (60–200 ms). The 5G configuration (red) operated over 6 km, achieving 271 ms mean latency (206–307 ms). In both cases, control swap time between local and remote surgeons was under 3 seconds.

3.9 Case Study: Réseau de Transport d'Électricité

Réseau de Transport d'Électricité (RTE) is France's independent transmission system operator, responsible for transmitting electricity from large-scale generation sources—nuclear, hydroelectric, wind, and solar—to regional distribution networks serving nearly 67 million residents. It also facilitates substantial cross-border exchanges, combining technical and economic management in a single operational remit.

The network encompasses approximately 100,000 kilometres of high-voltage transmission lines and produces roughly 490–500 TWh of electricity annually (recent years). Power is transported at 400 kV along main arteries to minimise resistive losses; approximately 2,200 transforming substations step it down for regional distribution and ultimately for consumption.

The system employs a large-scale distributed, hierarchical and redundant topology. Each substation uses so-called Intelligent Electronic Devices (IEDs). An IED is a microprocessor-based controller that integrates functions such as protection, control, monitoring and communication within the substation environment. These devices continuously measure electrical quantities—such as current, voltage and frequency—and execute protective relaying functions. For example, when an IED detects that parameters exceed predetermined thresholds, it can initiate actions like opening or closing circuit breakers to isolate

faults, re-route power flows, or adjust transformer tap settings, thereby preserving the stability of the system. The inherent intelligence and rapid response time of IEDs are essential for handling unforeseen contingencies, reducing downtime and minimising the risk of widespread outages. IEDs allow for an element of distributed autonomy, where individual subsystems can make decisions to even out demand and respond to localised failures without the need for external intervention.

Local redundancy is implemented through the deployment of multiple IEDs, often on parallel communication channels within a substation. This ensures that if one device fails, another can assume the necessary control and protective functions without interruption. Additionally, local control systems in substations are designed to monitor a range of conditions—including overcurrent, undervoltage, or frequency deviations—and to respond autonomously. Actions taken at the local level may include the tripping of specific feeders, reconfiguration of network topology, and activation of backup generation or storage systems. These measures are typically executed in real time in response to transient faults, sustained overloads, or sudden changes in demand, contributing significantly to the overall robustness and fault tolerance of the grid.

IEDs detect common fault classes—overcurrent, high impedance, voltage drop, phase imbalance, and surge—and respond autonomously by isolating affected circuits, reconfiguring network topology, or adjusting transformer settings, without waiting for instruction from a central controller.

IEC 61850 defines the architecture, communication protocols, and data models for substation automation. It standardises high-speed data exchange between IEDs and control systems—using mechanisms such as GOOSE messaging and Sampled Values—enabling real-time communication within the substation and reducing engineering complexity through standardised configuration language (SCL) files.

The French system uses an industrial standard known as Supervisory Control and Data Acquisition (SCADA) for the monitoring and control of electrical systems. It provides a platform through which aggregated information from IEDs is communicated to control centres. SCADA systems receive data on various operational parameters, such as voltage levels, current flows, frequency, equipment status, fault indicators and alarm signals. These measurements, along with real-time diagnostics and status reports, are transmitted over redundant communication networks.

Redundancy is a critical feature of substations, ensuring that even if one component fails or communication paths are disrupted, alternative routes can maintain the flow of information. IEDs, often installed in structurally redundant configurations, continuously monitor the substation's operational parameters and are programmed to initiate protective measures by signalling faults or abnormal conditions. The SCADA system interprets these signals to trigger automated responses, such as isolating faulted segments or redistributing loads, whilst simultaneously alerting operators for manual intervention if nec-

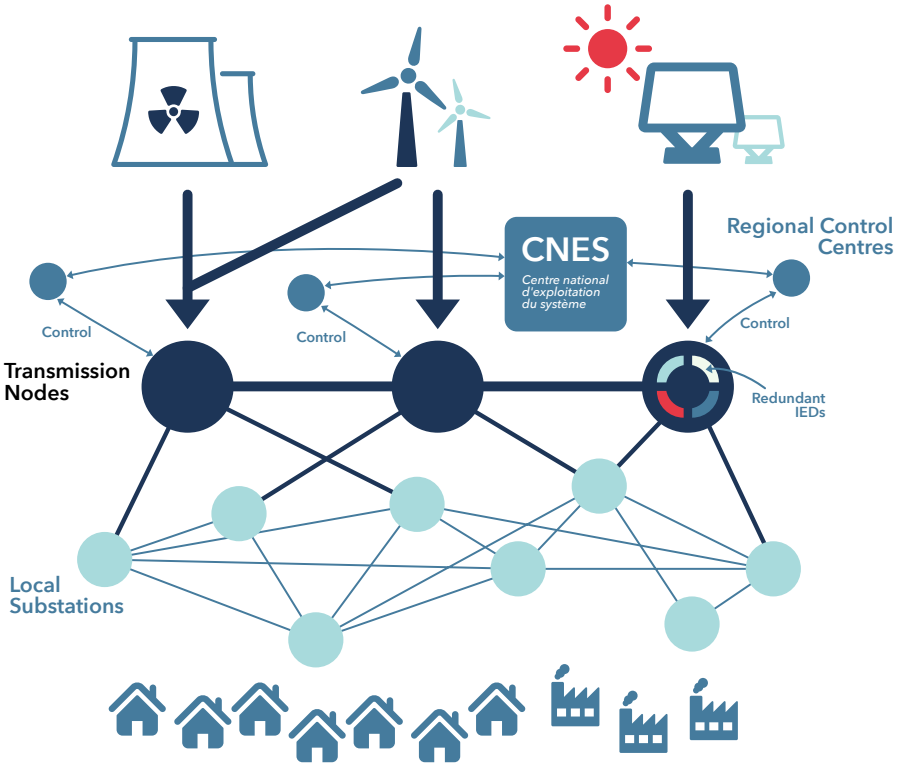


Figure 3.11: An overview of the French power transmission system. At the top level, sources of power including nuclear, wind and solar feed into the system. Large-scale substations act as transmission nodes and distribute power at high voltages over large distances and to local substations. A central control centre, CNES, manages the load across the whole system and works with seven additional regional control centres to ensure reliable supply across the country. The whole system is interconnected in a mesh-style topology, with capacity for alternate routing in the case of a subsystem failure.

essary. Through a combination of hardware redundancy, dual communication channels and integrated fault-detection algorithms, the overall system is engineered to provide a reliable, resilient and timely response to both transient and permanent faults, preserving system stability and ensuring the continuous operation of the grid.

For this thesis, the RTE case contributes a different lesson from the A320 and Kangduo examples. It shows that real-world reliability is often achieved not by a single intelligent controller, but by layered autonomy: local devices handle faults quickly, regional structures coordinate wider responses, and central systems maintain system-wide balance. That pattern matters for reinforcement learning because it suggests that distributed control should be designed around clearly separated responsibilities, bounded communication roles, and explicit fallback structure rather than around a single monolithic policy. In that respect, the grid looks less like one optimiser and more like a managed collection of specialised units operating at different timescales. This pattern—local fast responses at the IED level, slower regional coordination, and global optimisation at SCADA—directly parallels the timescale decomposition in Chapter 5, where distributed policy units operate on commitment bounds that enforce analogous temporal separation.

Chapter 4

Works

Abstract

Reinforcement learning has proven itself in a wide array of simulated domains, yet deploying it in real-world systems exposes a cluster of persistent obstacles: sample scarcity, system constraints, partial observability, reward misspecification, the need for offline training, interpretability requirements, high-dimensional spaces, and—most pertinently for distributed systems—latency and actuator delays. This chapter surveys those challenges, illustrates them through three case studies in sepsis treatment, robotic manipulation, and telesurgery, and synthesises the recurring deployment gaps that motivated the technical contributions developed in subsequent chapters: policy graphs for interpretable modular control, EnvCraft for generalisation benchmarking, MiniConv for edge-optimised inference, and CALF for communication-aware training.

4.1 Foundations

Dulac-Arnold et al. [44] discuss several of the challenges involved in real-world applications of RL. Namely:

1. Being able to learn on live systems from limited samples.
2. Reasoning about system constraints that should never or rarely be violated.
3. Interacting with systems that are partially observable, which can alternatively be viewed as systems that are non-stationary or stochastic.
4. Learning from multi-objective or poorly specified reward functions.

5. Training offline from the fixed logs of an external behaviour policy.
6. Providing system operators with explainable policies.
7. Learning and acting in high-dimensional state and action spaces.
8. Being able to provide actions quickly, especially for systems requiring low latencies.
9. Dealing with unknown and potentially large delays in the system actuators, sensors, or rewards.

4.1.1 Limited Samples

Learning high quality policies using only a limited number of experiences is a common problem in RL, often referred to as a problem of *sample efficiency*. The problem is most acute when training policies in the real world, since the acquisition of experiences can often be costly. In robotics, steps can typically only be carried out sequentially and in real time, so the collection of large amounts of training data can take a prohibitively long time. This limitation necessitates algorithms that can generalise effectively from sparse interactions, compared to traditional simulation-based RL where samples are abundant and cost-free.

Research has tackled this challenge through several approaches. Model-based RL blends learned dynamics with model-free updates to reduce real-interaction requirements [45], whilst meta-learning algorithms such as MAML [46] accelerate adaptation to new tasks by leveraging prior experience. Off-policy methods further improve efficiency by reusing past transitions via experience replay [12]. Later chapters return to this pressure through reusable policy units and environment-generation pipelines intended to extract more value from limited real interaction.

4.1.2 System Constraints

System constraints in RL, such as safety boundaries or operational limits, are often a critical requirement in real-world applications like autonomous driving, robotics, and healthcare. Unlike unconstrained settings where exploration is unbounded, real-world systems demand that agents avoid violating rules—such as the collision avoidance in vehicles or dosage limits in medicine—during both learning and deployment. This challenge requires balancing exploration with adherence to hard or soft constraints, often conflicting with reward maximisation objectives.

In practice, these constraints are often intertwined with where computation is placed. For example, Neurosurgeon [47] partitions deep neural network inference between mobile devices and the cloud, explicitly optimising for end-to-end latency and energy by deciding which layers run where. This kind of computation offloading highlights that latency is

not merely an algorithmic property but a system-level design choice: a policy's physical location and the communication topology can drastically change the effective feedback delay observed by an RL agent.

Constrained optimisation, as in Achiam *et al.*'s CPO [48], formulates RL to maximise rewards within explicit cost limits, whilst shielding [49] employs runtime monitors to block constraint violations outright. Both approaches trade some learning efficiency for a safety guarantee, a tension that reappears in the thesis through constrained routing, bounded commitment, and explicit fallback structure.

Large-scale RL systems make similar trade-offs. Sample Factory [50] demonstrates a single-machine architecture capable of exceeding 100,000 environment frames per second by aggressively parallelising simulation and learning, whilst Isaac Gym [51] keeps physics and policies on the GPU to avoid CPU–GPU communication bottlenecks. Both systems illustrate that choices about simulation architecture can introduce staleness and latency between data collection and policy updates, even when communication occurs on a single physical node.

4.1.3 Partial Observability

Partial observability, where agents cannot fully perceive the environment's state, is a pervasive challenge in real-world RL. A robotic manipulator working in clutter must infer object pose from occluded views and sensor history; a clinical policy must act on noisy, missing, or delayed measurements rather than complete physiological data. These situations can be modelled formally as POMDPs, requiring agents to maintain and act upon belief states rather than exact state estimates.

Recurrent networks [52] approximate belief states by tracking observation sequences over time, whilst model-based approaches such as Hafner *et al.*'s latent dynamics model [53] learn predictive world models from which hidden states can be inferred. The issue matters directly for this thesis because later systems must cope with stale observations, sparse interface cues, and network-mediated state information rather than perfectly exposed simulator state.

4.1.4 Reward Functions

Multi-objective or poorly specified reward functions challenge RL by introducing conflicting goals or ambiguous success criteria. In navigation robotics, objectives might be specified as a combination of path accuracy, speed, duration or other measures. In the domain of healthcare, different optimisation criteria routinely conflict; reduced patient mortality might result in a higher financial cost or a reduced infection rate might result in reduced patient satisfaction. Agents must balance these objectives or infer intended

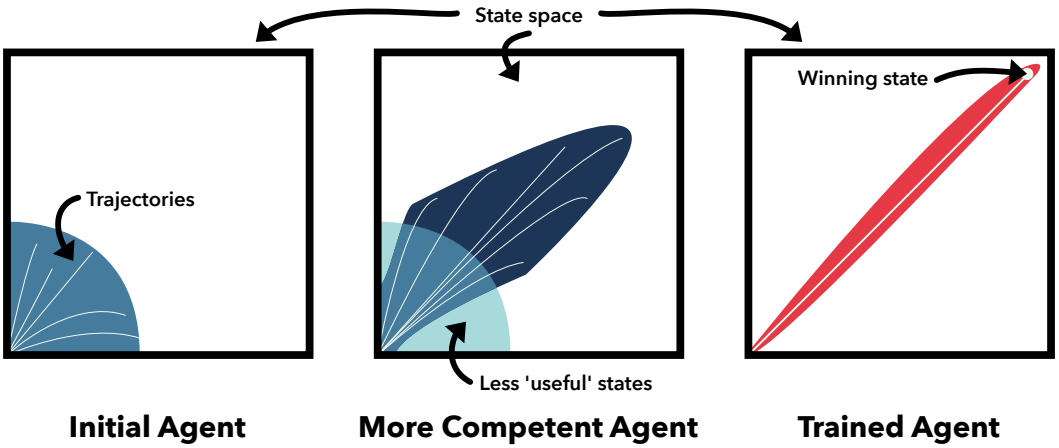


Figure 4.1: Online training is shown by visualising state spaces in 2D. Initially, a collection of easy-to-achieve states is seen by the agent and used for training. By using reinforcement, through the rewards achieved at different states, less useful states are no longer visited and harder-to-achieve states are discovered. As the harder-to-achieve states become more routinely explored, the policy is further able to achieve more difficult states. Over time, the policy explores a narrower set of states and is able to achieve more complex combinations of actions.

rewards, as misaligned or vague specifications can lead to suboptimal or unintended behaviours. This complexity requires robust reward design or adaptive learning to align policies with real-world intent.

The literature offers a range of approaches to this issue. Multi-objective RL employs *scalarisation* or *Pareto optimisation* to balance goals [54], inverse RL infers rewards from expert demonstrations [55], and reward shaping adjusts rewards to guide behaviour [56]—though misdesign in any case risks unintended outcomes. For the present thesis, this is one reason to prefer architectures that expose intermediate decisions and operational traces rather than leaving all reward interpretation buried inside a monolithic policy.

4.1.5 Offline Training

As shown in Figure 4.1, online training uses *bootstrapping*: a random policy first explores easily accessible states; as those experiences improve the policy, it reaches progressively harder states, creating a virtuous cycle of data collection and learning. Training RL offline loses this feedback loop—fixed logs of experiences are used to train a policy that, *were it to have acted in the environment*, would maximise reward. An iterative variant, known as *batch* RL, alternates experience collection and offline training; Figure 4.2 illustrates all three regimes.

Offline training is especially prevalent in healthcare, finance, and robotics, where real-world data collection is expensive and online exploration is unsafe or impractical. Recent surveys, such as Fu *et al.* (2023) [57], show how widely offline learning is now used

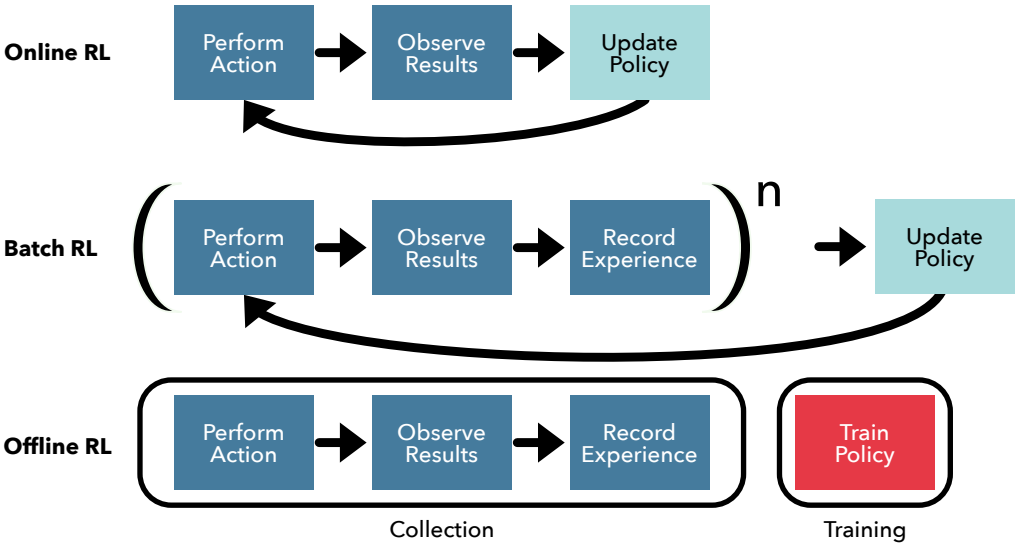


Figure 4.2: Online RL updates the policy after encountering new experiences. Batch RL updates the policy for every batch of experiences. Offline RL trains on pre-existing experience data without those experiences reflecting feedback from the progressively trained model.

in robotics, healthcare, recommender systems, and autonomous driving. Foundational methods such as *Batch-Constrained Q-Learning* (BCQ) [58] helped establish the core problem of *distributional shift*: a learned policy may choose actions that are poorly supported by the fixed dataset on which it was trained. Benchmarks such as *D4RL* [59] subsequently gave the field a common evaluation language, and methods such as *Conservative Q-Learning* (CQL) [60] were developed specifically to reduce overestimation on out-of-distribution actions.

In robotics, offline RL is attractive because collecting real interaction data is expensive and slow. Chen *et al.*'s *Batch Exploration with Examples* (BEE) [61] addresses this by using a small amount of human guidance to steer exploratory data collection towards task-relevant regions before offline training. Zentner *et al.* [62] likewise show that transfer structure across related tasks can reduce the amount of new data required. These examples matter because they show that offline RL is most useful when the data-collection process itself is engineered rather than treated as an afterthought.

Beyond robotics, offline RL identifies high-risk treatment patterns in healthcare without trial-and-error on live patients [63], and supports policy learning from pre-collected driving logs where online exploration would be unsafe [64].

Methodologically, the field now spans straightforward batch adaptations of value learning [65], off-policy evaluation techniques based on importance sampling [66], conservative regularisation [60], model-based variants such as MOReL [67], and sequence-modelling approaches such as Decision Transformer [68]. Fujimoto *et al.* further showed that a relatively simple modification of TD3—TD3+BC, which adds a behaviour-cloning

term and state normalisation—can achieve competitive D4RL results with relatively low implementation complexity [69]. The common limitation remains data quality: offline RL is powerful when logs have adequate coverage and sensible support, but brittle when the dataset omits critical states, actions, or failure modes. This is why later chapters emphasise inspectable execution structure and controlled proxy environments: in deployment settings, the dataset is rarely rich enough to let opacity be harmless.

4.1.6 Explainable Policies

Explainable policies are essential in real-world RL for trust and accountability, particularly in life-critical applications like healthcare and autonomous driving, where opaque decisions undermine adoption. In deployment, it is not enough to know that a policy works on average: operators need to inspect why a recommendation was made, what evidence it relied upon, and how the system behaved when it failed. Interpretable model architectures [70] and post-hoc explanation methods such as LIME [71] address part of this need, as do saliency visualisations [72].

A useful distinction for the chapters that follow is between *feature-level explanation* and *execution-level explanation*: saliency maps may show what a policy attended to, but deployed systems also require readable traces of which unit acted, when control changed hands, and what fallback occurred. That need recurs in the sepsis and telesurgery case studies below, and later motivates the modular execution structures developed in this thesis.

4.1.7 High-dimensional State and Action Spaces

High-dimensional state and action spaces strain traditional algorithms designed for low-dimensional, discrete settings. States may include raw sensory data such as images or multi-variable financial indicators, whilst actions can span continuous ranges such as motor controls, exponentially increasing computational demands and sample requirements.

Deep RL has been the primary response: DQN [12] demonstrated that discrete action control from images is tractable, whilst DDPG [73] extended this to continuous control. These results motivate the thesis's focus on compact encoders, conditional computation, and modular decomposition rather than ever-larger monoliths.

4.1.8 Latency

Low-latency action provision is essential in real-world RL systems like robotics, autonomous driving, and high-frequency trading, where delays can compromise both safety and efficacy. Computing actions with low latency—in the order of milliseconds—requires balancing policy complexity with execution speed, a departure from offline RL where

latency is less critical. This challenge requires efficient algorithms and infrastructure to ensure real-time responsiveness in dynamic environments.

Three recurring responses appear in the latency literature. *Policy distillation* compresses large models into smaller ones that can act more quickly at deployment. *Hardware acceleration* uses GPUs, TPUs, or specialist inference hardware to reduce wall-clock decision time. *Real-time planning* trades precomputed reactive behaviour for structured online search in domains where the planning horizon remains manageable.

Beyond purely algorithmic techniques, recent work has begun to treat delay as a first-class design parameter in deployment settings such as teleoperation. Bataduwaarachchi (2024) [74] proposes deterministic delay-aware reinforcement learning for teleoperated robotic systems, explicitly modelling end-to-end communication delays between operator, agent and environment. By adjusting how observations and actions are scheduled, these methods show that system-level design and delay-aware learning rules can be combined to maintain control performance under realistic network conditions.

The system-level techniques discussed in the System Constraints subsection—collaborative cloud–edge inference [47] and high-throughput simulators [50, 51]—further underscore that where and how computation is performed is inseparable from latency considerations in real-world RL. That observation leads directly to later chapters on compact edge models, commitment-bounded policy graphs, and network-aware training.

4.1.9 Dealing with Delays

Delays in actuators, sensors, or rewards disrupt the immediate feedback assumption central to traditional RL, complicating policy optimisation in real-world settings. Such delay—whether from mechanical lags in robotics, network latency in distributed systems, or delayed physiological responses in healthcare—introduce temporal misalignment between actions and their consequences, undermining standard Markovian assumptions. This challenge is particularly acute in systems where delays are variable or unknown, requiring RL agents to adapt dynamically to maintain performance.

MDPs that include a delay have been the subject of research interest for some time, but are now attracting renewed interest as RL is applied to real-world problems. Work by Brooks and Leondes (1972) [75] first discusses the issue of so-called 'state-information lag' in which the effect of actions is only seen after one timestep. Further early theoretical results involving MDPs with small constant delays are presented by Kim (1985) [76], Kim and Jeong (1987) [77], Altman and Nain (1992) [78] and Bander and White (1999) [79]. Similar problems have also been considered in the context of Dynamic Programming [80] and congestion control in high-speed networks [81].

Recent work extends these formulations to deep reinforcement learning with random, time-varying delays. Bouteiller *et al.* (2021) [82] analyse environments with stochastic ac-

tion and observation delays and introduce Delay-Correcting Actor–Critic (DCAC), which relabels trajectories in hindsight so that multi-step off-policy value estimates remain correct. Wang *et al.* (2024) [83] similarly formalise signal delay in continuous-control tasks and propose delay-aware actor–critic variants that achieve performance close to non-delayed baselines by carefully correcting for the misalignment between actions, observations and rewards.

Katsikopoulos and Engelbrecht (2003) [84] observe that delays in action execution ("action delay") and delays in state observation ("observation delay") pose equivalent problems from the position of the agent. They discuss formalisations for the Constant Delayed MDP (CDMDP) and the Stochastic Delayed MDP (SDMDP) and show how both can be reduced to problems dealing only with a single constructed MDPs. This result is important since it shows how the problem of finding an optimal policy for delayed MDPs can be solved using RL and gives us an indication of the increased complexity involved in optimally solving each problem in the general case.

Using these formulations, Katsikopoulos and Engelbrecht (2003) [84] show that the problem of finding optimal policies to CDMDPs is *NP-Hard*. Trivially, this is also true for solving SDMDPs. The implication of this result is that the development of an algorithm to solve delayed MDPs problem in a way which is *computationally feasible* [85] is extremely unlikely [86]. In line with this finding, some authors provide concrete examples of where heuristic-driven techniques are necessarily sub-optimal [87].

The effects of delays on the performance of naively applying existing algorithms has also been quantified. The performance of IMPALA [88] on a delayed environment degrades monotonically with the length of the delay [89]. Implementing a *waiting* agent, which simply waits for the delay to elapse before acting has also shown to perform poorly [90]. The more recent algorithms of Bouteiller *et al.* (2021) [82] and Wang *et al.* (2024) [83] can be interpreted as principled alternatives to such naive strategies: rather than waiting, they reconstruct or relabel the effective sequence of state–action–reward tuples, preserving the Markov structure needed by standard actor–critic methods whilst explicitly accounting for delayed execution.

A parallel line of work in networked control systems (NCS) studies similar phenomena from a control-theoretic perspective. Hespanha *et al.* (2007) [91] survey results on stability and performance of feedback loops in which sensors and actuators communicate over shared, lossy networks. This literature emphasises how packet loss, bounded or unbounded delays, and scheduling policies interact with closed-loop stability—issues that are increasingly relevant as RL controllers are deployed over the same kinds of shared communication infrastructure.

Several authors propose algorithmic solutions to MDPs with constant delays. Walsh *et al.* (2007) [90] introduce Model-Based Simulation (MBS) which uses a model to predict the most likely underlying (unobserved) MDP state and use the result as an input to

an RL training algorithm. Schuitema *et al.* (2010) [92] introduce modifications to the SARSA [93] and Deep-Q algorithms [12] to account for a constant known delay.

Firoiu *et al.* (2018) [89] revisit the technique of using a predictive model to account for delay. They implement a human-like predictive model using a GRU [94] and show how doing so significantly improves performance on the game *Super Smash Bros.* However, the success of this approach assumes that the state representation is semantically meaningful, which may not be the case in end-to-end systems.

Subsequent work has found some success in training RL algorithms using recent action buffers [95] and simple state prediction [96]. Liotet *et al.* (2021) [87] train a transformer network to generate a *belief* representation as a function of previous states and actions and train RL algorithms on this representation as normal.

One particularly impressive approach uses imitation learning to train agents to copy an expert trained on the non-delayed MDP [97]. However, it assumes knowledge of an underlying non-delayed environment which may not be present in most real-world scenarios.

A related work, addressing the problem of stochastic observation delays in the operation of a PD controller, provides discussion of the real-world problems faced in the control of devices operated at a distance such as medical and space equipment [98].

Almost all of the existing work on training policies on delayed MDPs considers only constant delays, specifically of *known* value. This is an assumption unlikely to hold in many real-world systems [98]. Many of the methods that do train on environments with stochastic delay still rely on assumptions that may fail in practice, such as knowing a small upper bound on the maximum delay [95]. Even more recent delay-correcting deep RL algorithms [82, 83] typically assume centralised training with full access to delay statistics and relatively clean interfaces between sensing, actuation, and computation, whereas real deployments must cope with heterogeneous hardware, network-induced variability, and partial observability on top of latency.

Furthermore, almost no existing work considers the difficult problem of *non-integer* delays in which the delay period may elapse *between* two MDP time steps. Schuitema *et al.* (2010) consider this problem using linear combinations of actions. Liotet *et al.* (2022) propose that some[1] non-integer delays may be treated as the combination of two *interleaved* MDPs.

Existing work has shown that the problem of delays in MDPs is provably *hard* and that only heuristically guided approximations are currently available. Despite this, some methods perform well under ideal conditions where delays are constant and known. There is still a long way to go: non-integer delays and stochastic delays remain largely unexplored, despite their relevance in real-world settings. Furthermore, there is no standard-

[1]The work considers two interleaved MDPs, allowing for delays to elapse either at the start of a time step, or at one single, predetermined interval within it.

ised methodology for training agents on delayed versions of environments, and current work reflects this by often demonstrating results on only one or a very small number of evaluation environments. The lack of a reusable systems methodology is one of the clearest motivations for the CALF infrastructure developed later in the thesis.

Similar issues arise in multi-agent settings, where agents must communicate over shared, noisy channels. Mao *et al.* (2020) [99] study multi-agent communication under limited bandwidth, introducing a gating mechanism that prunes redundant messages to respect communication budgets. Chen *et al.* (2020) [100] formalise Delay-Aware Markov Games and propose algorithms that mitigate the impact of action and observation delays across multiple agents. These works reinforce the view that delays and communication constraints are structural properties of many real-world control problems, not just incidental details of individual deployments.

4.2 Applications

4.2.1 Robotics

RL has been physically deployed across manipulation, locomotion, and navigation. Sim-to-real transfer is the central challenge: domain randomisation [101] and dynamics randomisation [102] train policies in simulation under a distribution over physical parameters so that they generalise to unknown real dynamics. Tan *et al.*'s quadruped system [103] provides a concrete example of treating latency as a first-class simulator design parameter: actuator latency is explicitly modelled and randomised alongside physical properties, so that the deployed policy is already adapted to realistic feedback delays.

4.2.2 Healthcare

Healthcare applications of RL confront stringent constraints: patient safety precludes exploratory learning on live subjects, regulatory requirements demand interpretability, and clinical datasets are often incomplete or biased by historical treatment protocols. Despite these obstacles, RL has been applied to treatment optimisation, drug dosing, and resource allocation.

Sepsis management has received significant attention, with policies trained offline on intensive care datasets to optimise fluid and vasopressor administration [104]. Such systems promise personalised treatment but face deployment barriers: clinicians require transparent decision traces, yet learned policies typically provide opaque recommendations. Section 4.3.1 examines this case in detail. Beyond sepsis, RL has been explored for chemotherapy scheduling [105], insulin delivery in diabetes management [106], and ventilator weaning protocols [107]. These applications share a common challenge: offline

learning from historical data introduces distributional shift, where policies encounter states absent from training logs, potentially yielding unsafe actions.

The requirement for offline learning stems from practical and ethical constraints. Randomised trials are expensive and slow; observational data is abundant but reflects clinician behaviour rather than optimal policy. Methods like Conservative Q-Learning address this by penalising out-of-distribution actions [60], whilst batch-constrained approaches prevent policy divergence from demonstrated behaviour [69]. However, conservatism trades safety for performance: policies may underperform human experts by avoiding beneficial but rarely-observed actions. Interpretability remains the critical deployment gap. Clinicians will not adopt systems that cannot explain why withholding treatment is recommended for a deteriorating patient, regardless of aggregate performance metrics.

4.2.3 Autonomous Systems

Autonomous vehicles represent RL's highest-visibility deployment domain, with substantial industry investment in perception, planning, and control systems. End-to-end learning approaches train policies directly from sensor inputs to control outputs, bypassing hand-engineered perception pipelines [108]. Whilst compelling in simulation, such systems confront severe generalisation challenges: training distributions cannot enumerate the long-tail of edge cases encountered in deployment.

Waymo's autonomous vehicles employ layered architectures combining learned perception with rule-based planners, reflecting pragmatic deployment constraints [109]. Perception failures—misclassified pedestrians, undetected obstacles, degraded sensor performance in adverse weather—require failsafe mechanisms and human oversight. RL has been applied to specific subproblems: lane-keeping, adaptive cruise control, and parking manoeuvres, where constrained operational domains permit reliable learning.

Network partitions and communication failures introduce additional failure modes. Vehicle-to-infrastructure systems assume reliable connectivity, yet cellular networks exhibit variable latency and packet loss. Distributed decision-making under network uncertainty motivates the communication-aware training methodology developed in Chapter 8. Beyond ground vehicles, unmanned aerial systems confront similar challenges: long-range operation requires tolerating communication delays, whilst safety-critical manoeuvres demand low-latency response. The Chapter 3 case studies of the Kangduo surgical robot and distributed power grids illustrate how engineered systems manage latency through explicit handover semantics and hierarchical control—principles applicable to autonomous vehicle fleets coordinating under network constraints.

4.2.4 Finance and Industrial Control

RL has been explored for algorithmic trading, portfolio optimisation, and market-making, where high-dimensional action spaces and non-stationary dynamics challenge traditional methods. High-frequency trading systems require sub-millisecond decision latency, constraining policy complexity and favouring compact representations [110]. Sample efficiency is critical: financial markets permit no exploratory losses, necessitating offline training on historical data with careful validation under realistic market conditions.

Industrial process control presents complementary challenges. DeepMind's data centre cooling system achieved substantial energy savings through learned control policies [111], demonstrating RL's applicability to complex multi-variate optimisation. Unlike healthcare or autonomous driving, industrial settings permit controlled experimentation: policies can be validated in simulation or shadow mode before deployment, mitigating safety risks.

4.3 Case Studies

The following three case studies examine in depth the deployment constraints most directly relevant to this thesis's contributions.

4.3.1 Sepsis Treatment in ICU

Summary & Methodology

Sepsis is a life-threatening dysregulation of the immune response to infection; left untreated it progresses to septic shock and multi-organ failure. Standard clinical management centres on intravenous fluid administration and vasopressor therapy to restore haemodynamic stability, with clinician judgement determining the dose and timing of each intervention.

The study *The Artificial Intelligence Clinician Learns Optimal Treatment Strategies for Sepsis in Intensive Care* by Komorowski *et al.* [104] trains an RL policy offline using data from over 17,000 patients from the *Medical Information Mart for Intensive Care (MIMIC-III)* dataset. Patient state is encoded as follows:

- The current state of the patient is represented by 48 clinical variables (including vital signs, laboratory values, and treatment history).
- The state is further discretised using k-means++ clustering, resulting in a total of 750 states across all 17000 patients' states.
- State is measured in four hour intervals.

- Data variables with multiple measurements within a 4-hour time step are summarised by averaging (for example, with heart rate) or summing (in the case of urine output).
- Missing data is handled using a *sample-and-hold* approach, where missing values are carried forward from the most recent available measurement.
- Two absorbing states are defined for discharge and death.

Notably, states are assumed to be homogeneous within each cluster, meaning that variations within a cluster are not explicitly accounted for when making decisions. Additionally, unlike human clinicians, the choice to model this scenario as a first-order MDP means that the agent cannot take previous state directly into account when making treatment decisions. The mapping from actions to physiological responses also does not account for patient-specific pharmacokinetics or pharmacodynamics, treating the effect of each action as identical across all patients in the same state.

Actions are simplified as follows:

- The action space is discretised into a fixed set of 25 possible actions, representing combinations of intravenous (IV) fluid and vasopressor dosages.
- IV fluid and vasopressor doses are each categorised into five discrete bins, where the lowest bin represents no drug administration, and the remaining nonzero doses are divided into quartiles.
- Rarely observed treatment decisions (defined as those occurring fewer than five times in the dataset) are excluded from the action space, potentially limiting the exploration of less common but effective interventions.
- The AI Clinician is constrained to learning on actions observed in the dataset, meaning it cannot learn anything about novel treatment strategies beyond those previously administered by clinicians.

The researchers have chosen to consider only two treatment modalities (IV fluids and vasopressors), excluding other relevant interventions such as antibiotics, corticosteroids, or mechanical ventilation, which may be used by a human clinician. Additionally, the policy cannot consider adjustments to treatment frequency or infusion rates, only total dose administration in each 4-hour time step. The model also assumes that, beyond the effect represented in a patient's state encoding, past treatment decisions do not influence future ones; as long as the patient's state does not represent any ill effect, a very large cumulative dose of IV might be administered over several timesteps, beyond what a human clinician would typically allow.

The agent is validated on the *eICU Research Institute (eRI)* dataset, which includes data on over 79000 admissions. Offline evaluation is conducted using *importance sampling* and bootstrapping to compare the agent's decisions with that of real clinicians.

Results & Analysis

To assess policy performance, the study employs off-policy evaluation using weighted importance sampling (WIS) and bootstrapping. Across 500 trained models, the agent's policy appears to consistently outperform clinician policies, achieving a 95% confidence lower bound that exceeds the upper bound of clinician performance. However, the model is limited by the constraints of retrospective data and potential confounders in the highly discretised and relatively narrow state representation.

Jeter *et al.* [112] provide a rigorous critique. They point out that the AI model's transition dynamics were not adequately validated, leading to questionable treatment recommendations, and that the use of four-hour data aggregation bins obscured rapid patient deterioration. The agent's performance degraded significantly on the external validation dataset, suggesting poor generalisability. Most strikingly, the agent sometimes chose non-intervention as the optimal strategy when a patient's Mean Arterial Pressure dropped below the recommended threshold—learning to associate intervention with patients already in stable conditions rather than as a necessary response to deterioration. The critique emphasises the need for transparency, reproducibility, and careful evaluation before AI can be safely integrated into medical decision-making, and the absence of publicly available code made independent verification impossible. This case also illustrates the quality–latency trade-off shown in Figure 4.3: the most clinically meaningful decisions require expensive offline optimisation, not fast look-up.

4.3.2 Batch Exploration for Robotic Manipulation

Summary

Robotic manipulation in unstructured environments confronts a fundamental challenge: acquiring diverse training data without exhaustive human supervision. Random exploration in high-dimensional visual observation spaces is prohibitively sample-inefficient; task-specific demonstrations are expensive to collect at scale. Chen *et al.*'s *Batch Exploration with Examples* (BEE) framework [61] addresses this by using minimal human guidance to direct autonomous data collection, then training policies offline on the resulting dataset.

The approach targets vision-based manipulation tasks where the agent observes only pixel inputs and must learn control policies for object interaction. Unlike methods that require dense human demonstrations for every target task, BEE collects a single batch of

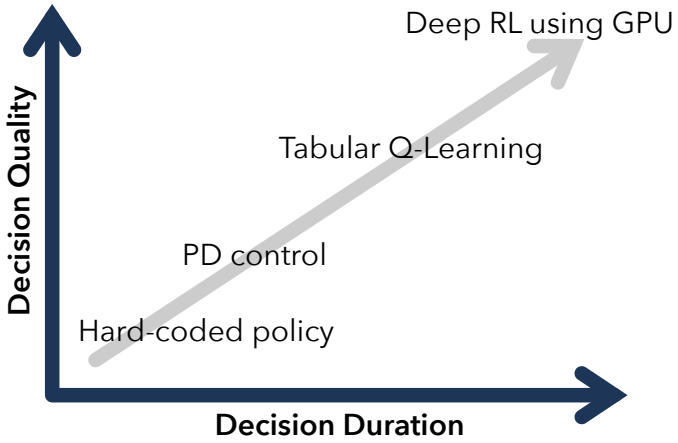


Figure 4.3: As decision quality improves, the amount of time taken to make the decision also usually increases. GPU-based RL requires the additional overheads involved in moving data between devices and usually involves larger, more complex models. Hard-coded lookup-based polices can respond quickly, but usually make poorer, less robust decisions.

exploratory data guided by a small set of example trajectories, then extracts task-specific policies through offline RL. This separates data collection (task-agnostic exploration) from policy learning (task-specific optimisation), enabling reuse of collected data across multiple downstream objectives.

Methodology

BEE operates in three phases. First, a relevance discriminator is trained on a small set (10–100) of human demonstrations to distinguish task-relevant states from random interactions, providing a reward signal that biases autonomous exploration towards useful regions without specifying the task goal. Second, model-based planning generates exploratory trajectories that visit relevant regions whilst maintaining diversity. Third, offline RL trains task-specific policies on the resulting dataset, using hindsight relabelling and conservative value estimation to handle distributional shift. In high-dimensional spaces, this guidance addresses a critical failure mode: random action sequences rarely produce meaningful object interactions, yet dense demonstrations are too expensive to collect at scale.

Impact

Experiments on vision-based pushing and grasping tasks demonstrate that BEE substantially outperforms baseline offline RL methods trained on randomly collected data. Policies trained on BEE datasets achieve success rates exceeding 70% on held-out object configurations, compared to <20% for random exploration baselines under matched data

budgets. Critically, BEE's data collection is task-agnostic: the same exploratory dataset supports training policies for multiple objectives (e.g. pushing objects to different target locations) without re-collecting demonstrations.

The work highlights a recurring deployment theme: sample efficiency through structured exploration. Pure offline RL assumes access to high-quality datasets; pure online RL assumes cheap interaction. Real robotic systems permit neither: data collection is expensive, yet available historical data may not cover task-relevant states. BEE's hybrid approach—autonomous exploration guided by minimal human input—provides a pragmatic middle ground. However, the method inherits offline RL's brittleness: policies fail when deployed states deviate from the training distribution. The relevance discriminator also introduces a potential failure mode: if human examples inadequately represent the task, exploration may focus on irrelevant behaviours.

This case study connects to subsequent thesis contributions. The need for sample-efficient skill acquisition motivates policy graphs' modular training interfaces (Chapter 5), where low-level units learn reusable primitives from simple feedback whilst higher-level components compose them into task-specific behaviours. BEE's reliance on offline learning underscores the value of diverse training distributions, motivating EnvCraft's environment generation methodology (Chapter 6). The vision-based observation space and associated computational demands exemplify the edge deployment challenges addressed through MiniConv's compact encoders (Chapter 7).

4.3.3 Telesurgery and Latency Predictability

The Kangduo KD-SR-01 telesurgical system, examined in detail in Chapter 3, provides the clearest evidence of the latency predictability principle. Experimental deployments over links of 80 km and 6 km established that consistent latency of 130–271 ms supports safe telesurgery, whilst higher jitter at equivalent mean delay causes positional error and surgeon confusion [43, 28, 29]. The system's architectural response—dedicated leased lines for bounded worst-case delay, dual-console redundancy for local fallback, and a sub-three-second handover mechanism—instantiates the principle that predictable moderate latency outperforms unpredictable low latency. Chapter 8 operationalises this insight through CALF's network-aware training, which exposes policies to realistic latency distributions during simulation, yielding robustness that zero-latency training cannot provide.

4.4 Synthesis: Recurring Deployment Challenges

The foundational challenges, application domains, and case studies examined above reveal persistent obstacles to real-world RL deployment. These obstacles transcend specific application areas, appearing across healthcare, robotics, autonomous systems, and in-

dustrial control. This section synthesises common themes and identifies deployment gaps that motivate the technical contributions in subsequent chapters.

4.4.1 Interpretability and Accountability

The sepsis treatment case study exemplifies a critical deployment barrier: clinicians will not adopt systems they cannot interpret. Komorowski *et al.*'s AI Clinician achieves superior aggregate performance metrics through offline RL, yet Jeter *et al.*'s critique reveals that the policy sometimes recommends withholding treatment for deteriorating patients without providing explanatory traces. Clinicians require decision rationales—which observations triggered which actions, and why—not merely confidence scores or aggregate survival rates.

This interpretability requirement extends beyond healthcare. Financial regulators demand audit trails for algorithmic trading decisions; industrial operators require explanations when process control policies deviate from established procedures; autonomous vehicles must justify emergency manoeuvres to accident investigators. Black-box policies, regardless of performance, fail to meet these accountability standards. Traditional RL produces monolithic neural networks mapping observations to actions with no intermediate structure; post-hoc explanation methods provide approximations but cannot guarantee faithful decision traces.

Chapter 5 addresses this through policy graphs: a modular architecture where decision-making decomposes into *units* communicating through explicit interfaces, with hard-routing ensuring deterministic call-and-return traces. Unlike ensemble methods or hierarchical RL, policy graphs provide *accountability by construction*: each decision is attributable to a specific unit, and execution paths are observable through routing tables, meeting the regulatory and clinical requirements that monolithic policies cannot satisfy.

4.4.2 Sample Efficiency and Offline Learning

All three case studies confront sample scarcity. Sepsis treatment cannot permit exploratory administration of harmful drugs; robotic manipulation systems cannot afford thousands of hours of real-world interaction; telesurgery systems must operate reliably from initial deployment. Consequently, real-world RL relies heavily on offline learning: training policies from fixed datasets collected under historical behaviour.

However, offline learning introduces distributional shift: policies encounter states absent from training data, yielding extrapolation errors that online learning avoids through bootstrapping. Conservative methods mitigate this by restricting policies to demonstrated behaviours, but conservatism sacrifices performance—policies cannot discover better-than-human strategies if constrained to imitate historical data. The BEE framework partially addresses this through guided exploration, collecting diverse data without

task-specific supervision, but still depends on the relevance discriminator adequately representing task requirements.

Policy graphs improve sample efficiency through modular training. Rather than learning monolithic end-to-end policies requiring comprehensive datasets, individual units learn narrow sub-tasks with simple reward signals. A vision encoder learns from self-supervised reconstruction; a low-level motor controller learns from position tracking errors; a high-level planner learns from sparse task completion. Each unit's training data requirements are modest compared to end-to-end alternatives, and units can be pre-trained on diverse tasks then composed for new objectives without full retraining. This compositional reuse reduces the sample burden that offline learning imposes.

4.4.3 Latency Predictability vs. Sporadic Low Latency

The telesurgery case study establishes a counterintuitive principle: *predictable moderate latency outperforms unpredictable low latency.* Fan *et al.*'s deployments succeed under consistent 130-270 ms delays but would fail under variable 0-500 ms latency with the same mean, because human operators (and, by extension, learned policies) can adapt to consistent delays but cannot compensate for unpredictable jitter.

This insight contradicts common RL training assumptions. Standard benchmarks execute policies in lockstep with simulation: action a_t immediately produces next state s_{t+1} with zero delay. Deployed systems exhibit variable latency: network communication, sensor processing, actuator dynamics, and policy inference all introduce delays that fluctuate based on computational load and network conditions. Policies trained under zero-latency assumptions cannot adapt to deployment latencies, whilst policies trained under realistic latency distributions learn compensatory strategies—predictive control, delayed action execution, or conservative behaviour when latency exceeds thresholds.

Chapter 8 operationalises this through CALF: Communication-Aware Learning Framework. Rather than training policies in idealised zero-latency simulation, CALF exposes agents to realistic network models during training, including variable transmission delays, packet loss, and bandwidth constraints. Policies learn to tolerate communication failures, execute time-critical components locally whilst offloading computation when network permits, and gracefully degrade when latency exceeds bounds. This yields robustness that zero-latency training cannot provide.

The foundations section's discussion of actuator delays reinforces this theme. Robotic systems exhibit non-integer, stochastic delays between commanded actions and physical effects. Existing work demonstrates that constant known delays can be handled heuristically, but variable delays remain largely unexplored despite their prevalence. The distributed power grid and telesurgery examples from Chapter 3 illustrate how deployed systems manage latency through architectural choices: explicit handover semantics, local

fallback mechanisms, and bounded worst-case guarantees. These design patterns inform policy graphs' deployment model: safety-critical units execute locally with deterministic latency, whilst optimisation-oriented units execute remotely and tolerate variable delays.

4.4.4 Generalisation Beyond Training Distributions

All surveyed application domains confront generalisation failures. Autonomous vehicles trained on sunny highway driving crash in snow; robotic policies optimised in simulation fail on real hardware; healthcare policies trained on one hospital's patient population underperform at institutions with different demographics. The sim-to-real gap exemplifies this: domain randomisation and dynamics randomisation improve transfer, but policies still encounter deployment states outside their training distribution.

The BEE case study demonstrates that task-agnostic exploration improves generalisation by collecting diverse data, but the approach still depends on the relevance discriminator identifying appropriate state coverage. If training environments inadequately represent deployment diversity, policies fail. This motivates systematic environment generation rather than manual dataset curation.

Chapter 6 addresses this through EnvCraft: a procedural environment generation system producing diverse task variants covering broad state distributions. Rather than training on fixed benchmark suites or manually designed levels, policies train on programmatically generated environments spanning parameter ranges, obstacle configurations, and reward structures. EnvCraft's diversity metrics quantify environment coverage, enabling principled evaluation: does the policy generalise to held-out environment parameters, or merely overfit to training instances?

4.4.5 Edge Deployment and Computational Constraints

The foundations section's discussion of system constraints highlights computational trade-offs: placing computation on-device reduces communication latency but constrains model capacity; offloading to cloud permits larger models but introduces network delays. Neurosurgeon's DNN partitioning exemplifies this, whilst Sample Factory and Isaac Gym demonstrate that even single-machine systems confront latency-throughput trade-offs based on simulation architecture.

Robotic manipulation and autonomous vehicles require real-time inference on resource-constrained hardware: mobile robots carry limited battery and compute; embedded automotive systems face strict power budgets; surgical robots demand deterministic latency incompatible with cloud offloading. These constraints necessitate compact policy representations compatible with edge deployment.

Chapter 7 addresses this through MiniConv: compact convolutional encoders enabling vision-based policies to execute on edge hardware. Rather than deploying large ResNet or

Vision Transformer encoders requiring GPU acceleration, MiniConv provides parameter-efficient architectures achieving competitive performance within embedded system constraints. This enables the deployment model that telesurgery and robotics require: time-critical perception and control execute locally, whilst high-level planning may offload to remote infrastructure when network permits.

Policy graphs integrate with edge deployment through distributed execution: different units deploy on different hardware based on computational requirements and latency tolerances. A lightweight MiniConv vision encoder executes on-device; a heavyweight world model executes remotely; routing logic determines active execution paths based on current network conditions. This mirrors the dual-console telesurgery architecture: the local operator maintains control when the remote connection degrades. The same modular structure also addresses multi-objective constraints: safety-critical units enforce hard limits whilst optimisation-oriented units pursue performance objectives, separating concerns that monolithic reward shaping conflates.

4.5 From Gaps to Contributions

Table 4.1 maps each recurring deployment gap identified in this chapter to the contributing chapter and the key technique it employs.

Table 4.1: Deployment gaps identified in this chapter and their corresponding thesis contributions.

Deployment Gap	**Contributing Chapter**	**Key Technique**
Interpretability and accountability	Chapter 5	Policy graphs: hard-rout
Sample efficiency and offline learning	Chapter 5	Modular unit training; c
Latency tolerance and network variability	Chapter 8	CALF: communication-a
Generalisation beyond training distributions	Chapter 6	EnvCraft: procedural en
Edge deployment and computation constraints	Chapter 7	MiniConv: compact conv
Physical realisation	Chapter 9	Hardware integration an

Real-world RL deployment requires simultaneously addressing interpretability, sample efficiency, latency tolerance, generalisation, and computational constraints. The technical contributions in Chapters 7 through 9 provide integrated solutions to these persistent challenges, grounded in the deployment gaps that this chapter's survey has identified.

Chapter 5

Effects

Abstract

Reinforcement learning has achieved notable successes with large models, yet scaling monolithic policies to long-horizon, high-dimensional environments remains challenging in practice. This chapter introduces *policy graphs*, an implementation-centric formulation for modular reinforcement learning in which callable *policy units* (skills/options) are composed as nodes in a directed graph and coordinated by explicit routing decisions with well-defined delegation and return semantics. The formulation unifies common hierarchical patterns whilst enabling practical modular training and deployment, including constrained transitions, heterogeneous unit implementations, and bounded commitment to reduce unstable switching. To connect the abstraction to deployment-relevant interaction regimes, we also introduce BrowserEnv and FilesEnv: lightweight proxy environments with simple, reproducible dynamics but complex, real-world-like interaction requirements. The chapter then develops two complementary construction routes for policy graphs: a teacher-guided synthesis pipeline that discovers candidate specialists from action-conditioned saliency in controlled MiniGrid tasks, and a hard-routing instantiation over a fixed pool of specialists, compared against soft mixture-of-experts baselines, in deployment-motivated domains. Together these studies address both sides of the construction problem: where specialist units come from, and how routing over those units can be stabilised in practice.

5.1 Introduction

In Chapter 3, we examined how real-world systems manage complexity through carefully engineered patterns of specialisation, hierarchy, and delegation. The Airbus A320

achieves reliable flight control by distributing responsibility across dedicated computers—Elevator and Aileron Computers (ELACs) for pitch and roll, Spoiler and Elevator Computers (SECs) for backup control, and Flight Control and Guidance Computers (FCGCs) for higher-level coordination—each operating within well-defined interfaces and constrained transition rules embodied in the aircraft's flight laws. The French power transmission network maintains grid stability through a three-tier hierarchy: local Intelligent Electronic Devices (IEDs) respond autonomously to immediate faults, regional substations coordinate load balancing, and a central control system (CNES) manages nationwide demand. The Kangduo surgical robot enables remote telesurgery by implementing explicit handover semantics between local and remote surgeons, with sub-three-second delegation transitions and robust fallback to local control when network conditions degrade. These systems share a common architecture: *specialised units with distinct responsibilities, coordinated through explicit delegation and return mechanisms, operating under hard constraints that ensure predictable, accountable behaviour.* These principles trace to Adam Smith's division of labour—the insight, elaborated in Chapter 2, that specialisation and coordination drive productivity.

This chapter proposes *policy graphs* as a formalism that distils the architectural patterns observed in Chapter 3 into an implementation-ready abstraction for modular reinforcement learning. A policy graph is a directed graph $G = (V, E)$ whose nodes are callable *policy units*—analogous to the A320's flight computers or the power grid's IEDs—and whose edges constrain permissible delegations, much as the A320's flight laws govern transitions between control modes. Execution follows explicit call-and-return semantics: at any time a single unit is active, and it may (i) act in the environment, (ii) delegate control to a permitted successor, or (iii) return control to its caller. This mirrors the surgical robot's dual-console handover, where control authority transfers cleanly between operators with unambiguous responsibility at each moment.

Policy graphs address three gaps left by existing hierarchical RL formulations. First, they provide *operational semantics* that are directly implementable: delegation is a first-class operation with defined preconditions (edge constraints), commitment bounds prevent unstable switching (analogous to the A320's phase-based cockpit communication rules), and call traces provide accountability (as required for debugging real systems). Second, they unify diverse hierarchical patterns—options, feudal hierarchies, manager–worker systems—within a single framework whilst enabling non-tree topologies that better reflect real-world redundancy and shared subskills, much as the A320's three hydraulic circuits provide overlapping coverage of critical actuators. Third, they expose explicit control points for deployment constraints: individual units can be trained, tested, swapped, or distributed across heterogeneous hardware independently, whilst routing decisions remain inspectable and constrained, addressing the transparency and modularity requirements identified in real-world automation systems.

In the environments that motivate this work—web browsing, file-system interaction, and similarly long-horizon, interface-driven domains—complexity arises from the need to compose many precise, low-level operations into coherent workflows. Monolithic end-to-end policies struggle in this regime: credit assignment becomes difficult under sparse rewards, training is unstable when perception and control are learned jointly, and inference cost remains constant even when only a small subset of behaviour is relevant. Policy graphs address these challenges by enabling specialisation (low-level units master recurring interaction primitives), coordination (a router sequences units to achieve long-horizon objectives), and conditional computation (only the active unit and router incur inference cost). These mechanisms mirror those that enable the A320 to operate safely with degraded systems, the power grid to isolate faults without cascading failures, and the surgical robot to maintain control authority during network handover.

Chapter 4 identified interpretability deficits and latency unpredictability as the most blocking deployment gaps, motivating policy graphs' hard-routing call traces and commitment bounds respectively.

This chapter has three core contributions:

1. **Policy graph formalism and training template** (Contribution 1): We formalise policy graphs as directed graphs of callable policy units with explicit execution semantics (call-and-return, commitment bounds, constrained edges), and present a generic training template that supports modular data collection and updates. Policy graphs serve both as a learning structure—enabling skill specialisation and providing explainable routing decisions—and as a deployment framework that allows units to be distributed across different physical locations and hardware types, enabling System 1 impulses to execute on low-power edge devices near actuators whilst System 2 reasoning runs on remote GPU clusters.

2. **Real-world proxy environments** (Contribution 2): This chapter introduces BrowserEnv and FilesEnv, evaluation settings that deliberately couple simple, controllable dynamics with interface complexity characteristic of real-world deployment. BrowserEnv is used directly in the hard-routing study reported here, whilst FilesEnv broadens the interface setting and provides an additional proxy environment for future evaluation.

3. **Two empirical construction routes** (Contribution 3): First, this chapter shows that a competent monolithic teacher can be converted into a compact policy graph by clustering action-conditioned saliency traces into candidate behavioural regimes and distilling regime-specific specialists plus a router. Second, it evaluates hard attention routing over a fixed pool of specialists, with soft mixture-of-experts routing as a comparator, showing how the same policy-graph execution semantics can be realised when the unit inventory is fixed in advance.

5.2 Background and Related Work

The policy graph formulation synthesises insights from hierarchical reinforcement learning, real-world system design, and human skill acquisition. Chapter 2 established that division of labour—the principle that enabled Adam Smith's pin workers to achieve 240-fold productivity improvements—applies equally to learned control: specialised units coordinated through explicit mechanisms outperform monolithic approaches. Chapter 3 demonstrated how real-world systems (the A320's flight computers, the power grid's hierarchical control, the surgical robot's dual-console handover) embody these principles through redundancy, constrained transitions, and accountable delegation. Chapter 4 identified the deployment challenges (interpretability, latency predictability, safety constraints) that existing RL systems struggle to address. This section reviews the technical foundations that policy graphs build upon, connecting established hierarchical RL methods to the architectural patterns observed in engineered systems and the deployment demands identified in real-world applications.

5.2.1 Hierarchical Reinforcement Learning

Hierarchical reinforcement learning decomposes complex tasks into temporally extended subproblems, allowing policies to operate at multiple levels of abstraction—a computational analogue of the division of labour in Smith's pin factory (Chapter 2). The options framework formalises callable subpolicies with initiation sets and termination conditions [22], whilst feudal reinforcement learning emphasises hierarchical goal-setting between manager and worker levels [24]. These methods often require careful design choices around termination, subgoal representations, and skill priors, and can struggle to provide an implementation-level interface that supports flexible composition, constrained transitions, and deployment-oriented modularity.

5.2.2 Modularity, Routing, and Conditional Computation

Modular architectures provide an alternative route to specialisation: rather than imposing a fixed hierarchy, they learn to route computation through a subset of available modules. Mixture-of-experts models exemplify this idea by using a gating mechanism to select which expert(s) process a given input, trading dense computation for conditional activation [113]. In RL, similar routing mechanisms can be used to select among specialised encoders or policies on a per-state basis. A key practical distinction is between *soft* routing, which combines multiple modules in a weighted mixture, and *hard* routing, which selects a single module at a time. Hard routing enables three deployment-critical properties: first, *simplicity*—exactly one unit is responsible at any moment, making behaviour predictable; second, *single-state efficiency*—a routing decision can dispatch a single state

to specific hardware without waiting for all units to complete processing; third, *physical distribution*—units can be separated across different locations and hardware types (low-power edge devices for reactive control, remote GPUs for compute-intensive reasoning) with routing determining which device becomes active. These properties align naturally with deployment constraints (latency, memory, and interpretability), but require explicit mechanisms to avoid collapse and unstable switching, which are central to the policy graph formulation.

5.2.3 Teacher-guided Decomposition and Distillation

A complementary line of work uses a strong teacher policy to guide the training of smaller or more structured students. Distillation transfers behaviour from teacher to student via imitation objectives (e.g., KL divergence between action distributions), optionally followed by RL fine-tuning [114]. In interactive settings, teacher-guided approaches are often paired with dataset aggregation methods, such as DAgger [115], that address covariate shift between expert and learner rollouts. For policy graphs, the central opportunity is to *decompose* the teacher's behaviour into reusable units with explicit interfaces and routing structure.

5.2.4 Motivation from Human Skill Acquisition

The motivation for modularity is not solely computational. Chapter 2 traced reward signals from ancient philosophy through modern neuroscience, culminating in Schultz's discovery that phasic dopamine spikes encode reward prediction error—the brain's mechanism for reinforcing successful actions and chunking them into reusable behavioural routines. Human skill learning exhibits a gradual progression from stimulus-driven responses to autonomous execution of behavioural chunks. Fitts and Posner's theory describes a transition from a cognitive stage (fragmented individual steps), through associative refinement (formation of chunks aided by dopamine reinforcement), to autonomous execution (refined chunks performed automatically with minimal conscious effort) [116, 117]. These perspectives motivate a training strategy in which low-level policy units acquire reliable primitives from simple feedback—analogous to dopamine-driven chunking in the associative stage—whilst higher-level components learn to compose these primitives into long-horizon behaviour, mirroring the transition to autonomous skilled performance.

Taken together, existing HRL, modular routing, and teacher-guided learning provide powerful building blocks. However, they do not yield a single formulation that is simultaneously expressive (graph topologies), operational (explicit execution semantics), and implementation-ready (interfaces, buffers, and deployment constraints). Policy graphs are intended to fill this gap.

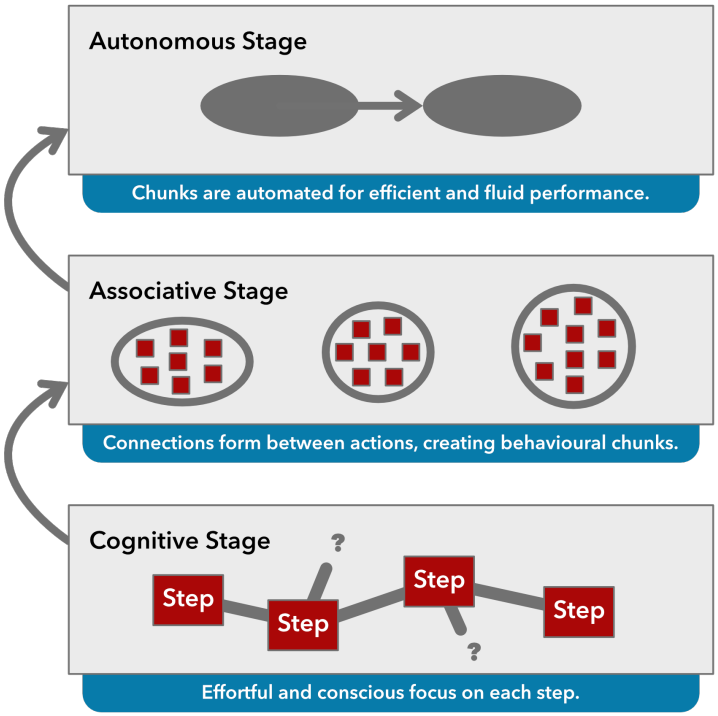


Figure 5.1: The diagram illustrates the stages of skill acquisition as proposed by Fitts and Posner (1967) [116]. In the cognitive stage, learners focus on fragmented individual steps. In the associative stage, repeated practice and feedback lead to the formation of behavioural chunks, aided by dopamine signalling, which reinforces successful action sequences (Schultz, 1998) [117]. By the autonomous stage, chunks are refined and executed automatically, enabling fluid and efficient performance with minimal conscious effort.

5.2.5 Policy Graphs as a Unifying and Generalising Framework

The policy graph formulation subsumes existing hierarchical RL approaches whilst addressing their practical deployment limitations. Options [22], feudal hierarchies [24], HAMs [26], and MAXQ [25] are each subsumed as special cases: options map to policy units with edge-encoded initiation sets; feudal managers map to routers; tree structures are relaxed to graphs that allow shared subskills and multiple callers. Soft MoE [113] is evaluated as a comparator in Section 5.7. Policy graphs unify these approaches whilst adding:

- **Explicit delegation semantics** (call-and-return) that make execution reproducible and debuggable.
- **Graph topologies** that generalise trees, enabling redundancy, shared subskills, and constrained transitions.
- **Commitment and termination bounds** that prevent unstable switching and provide worst-case guarantees, essential for real-world deployment.
- **Modular training interfaces** (unit-local buffers, call-level transitions) that support independent testing and swapping of components.
- **Hard routing semantics** that enable accountability, conditional computation, and physical distribution across heterogeneous hardware.

These properties are distilled from the architectural patterns of engineered systems examined in Chapter 3, providing a pathway from the operational clarity of those systems to the adaptability of learned policies.

5.3 Policy Graph Formulation

5.3.1 Definition

A *policy graph* is a directed graph $G = (V, E)$ whose nodes $v \in V$ are callable *policy units*. Each unit implements a policy π_v (and optionally a value function or critic) that maps its inputs to either an environment action or a routing decision. Units may be trained with standard RL algorithms, including value-based methods such as DQN [12] and policy-gradient methods [14].

Edges $(u \to v) \in E$ represent permitted delegations: from unit u, control may be transferred to unit v only if the corresponding edge is present. The routing decision can be implemented as an explicit *router* policy π_H (which selects the next unit), or embedded in the action space of the currently active unit; the formulation supports both, but the chapter emphasises hard-routing execution in which a single unit is active at any step.

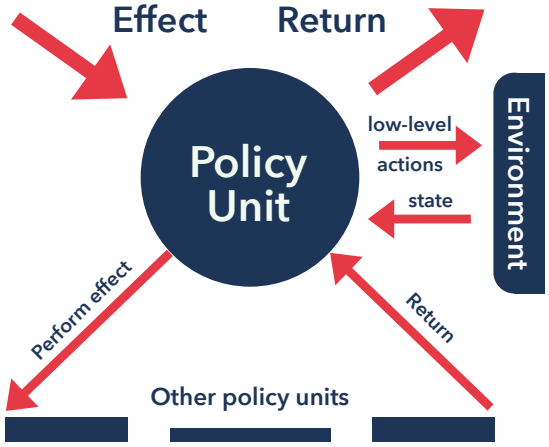


Figure 5.2: An illustration of a single policy unit as part of a larger policy graph. Policies have access to each of the actions in the environment's action space. Additionally, policies have pseudoactions corresponding to several effects and to move control flow to the previous policy unit.

Policy graphs require explicit interfaces to support modularity. At minimum, all units observe the current environment observation (or a shared embedding). In addition, transitions may carry structured information such as the caller identity, a compact memory state, or an "effect achieved" flag that indicates whether a delegated objective was satisfied. These interfaces are intentionally lightweight: they are meant to be implementable and debuggable, rather than maximally expressive.

5.3.2 Goals and Effects as Interface Primitives

Policy graphs do not require an explicit notion of subgoal. In many environments, however, it is convenient to label delegations using goal-like or effect-like primitives: higher-level components can delegate *what should be achieved* rather than *which low-level action should be taken next*. This section briefly formalises goals and effects as optional interface choices used in parts of this chapter.

Goal-conditioned environments

RL environments are commonly formalised as Markov Decision Processes (MDPs). An MDP is a structure $\text{MDP}(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ where:

- $\mathcal{S}$ is a set of states.
- $\mathcal{A}$ is a set of actions.
- $\mathcal{T}(s, s'|a) = \mathbb{P}(s_{t+1} = s'|a_t = a, s_t = s)$: The probability of a transition occurring between states s and s' when the agent takes action a.

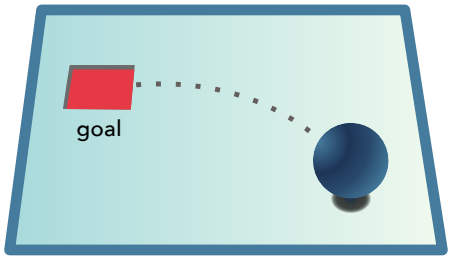


Figure 5.3: An example of a goal-conditioned environment. The agent controls a ball which can move in any direction in a 2D environment. The goal and state spaces are defined as $\mathcal{G} = \mathcal{S} = \mathbb{R}^2$. If $\textsc{Near}(s, g)$ is a predicate symbol which is true if, and only if the state s is within some defined distance of the goal g then $\textsc{Sat}(s, g) \iff \textsc{Near}(s, g)$.

- $\mathcal{R}(s, a, s') = \mathbb{E}(r_t | s_t = s, a_t = a, s_{t+1} = s')$: The expected reward gained when the system transitions from state s to s'.

Goal-conditioned formulations augment the MDP with a goal variable. A GMDP extends an MDP with a goal space $\mathcal{G}$ and a goal-satisfaction relation $\textsc{Sat} \subseteq \mathcal{S} \times \mathcal{G}$. Episodes are conditioned on a sampled goal $g \in \mathcal{G}$, and rewards may depend on whether the current state satisfies the selected goal.

Goal-conditioned reward functions

In a goal-conditioned setting, a common choice is a sparse reward that agrees with the satisfaction relation, for example $\mathcal{R}((s, a, s'), g) = 1$ if $\textsc{Sat}(s', g)$ and 0 otherwise (or the corresponding change-based variant when success is defined by reaching a newly satisfied state). Techniques such as HER exploit this structure by relabelling goals post hoc to extract learning signal from trajectories that do not achieve the originally sampled goal [20]. In policy graphs, goal labels can be used as part of the routing interface, but they are not required by the formulation.

Effects

Goals specify desired end-states, whereas actions specify immediate state transitions. For modular control, it is sometimes useful to define an intermediate primitive that represents a desired *change* relative to the state at which it is chosen. We call such a primitive an *effect*. Formally, an effect e can be represented as a relation on states, and is satisfied when the environment transitions from an origin state s_0 to a state that stands in relation e to s_0. Figure 5.4 illustrates several design choices for effect structure.

One way to train effect-conditioned behaviour is to use an origin-relative reward that fires when the effect becomes satisfied, as shown in Figure 5.5. In practice, the origin s_0 is carried as part of the unit interface when an effect is selected, allowing the unit to detect effect satisfaction relative to the origin.

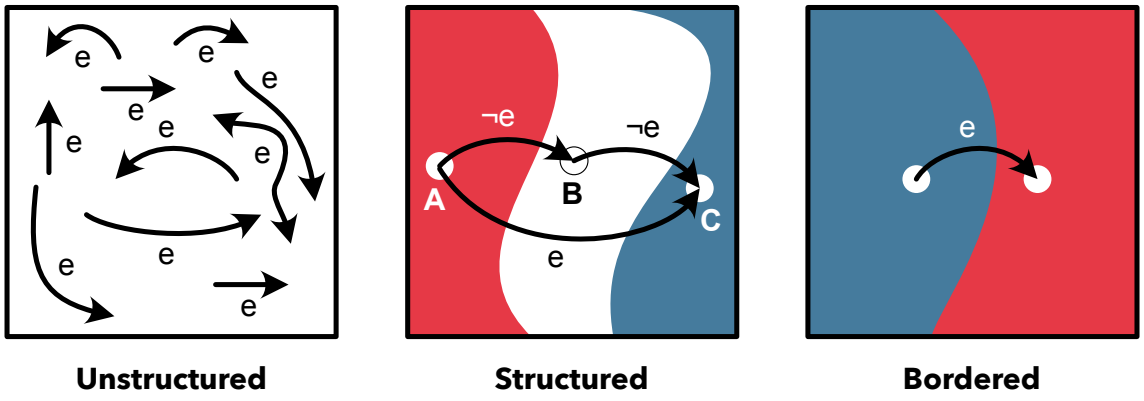


Figure 5.4: **A few examples of the ways in which effects can be designed within a state space.** In each case, the bordered square represents the state space of an environment. It is assumed that actions change the environment's state between adjacent locations within the square. In general, effects can represent an **unstructured** set of arbitrary ordered connections between elements of a state space. Alternatively, **structured** effects specify a collection of state subsets, between which all elements is connected by an effect. In the middle square, points **A**, **B** and **C** represent points in each of three such subsets represented by different colours. Structured effects have a correspondence to *goals*. For example, taking effect e from state **A** is equivalent to setting goal **C**. However, goals are independent of the effect origin. A specific form of structured effects are described as **bordered**. Bordered effects are simpler to learn than general effects, since the reward at each step is independent of the effect origin.

$$R([s, s_0, e], a, [s', s_0, e]) = \begin{cases} 1 & \text{if } s_0 \overset{e}{\not\to} s \wedge s_0 \overset{e}{\to} s' \\ 0 & \text{otherwise} \end{cases}$$

Figure 5.5: **The teleological reward function in e.** A reward of 1.0 is given when an action causes an effect to be satisfied relative to the effect origin.

In policy graphs, effects can be used as *routing labels*: selecting an effect is treated as selecting a particular unit (or class of units) expected to realise that effect, and satisfaction information can be returned to the caller via an interface flag. This is an optional modelling choice. The remainder of the chapter adopts the more general execution semantics in which units delegate to other units directly, with effects and goals available when they provide useful structure.

5.3.3 Execution Semantics

Policy graph execution is defined by an explicit control-flow mechanism. At any time, there is a single *active* unit u_t. Let $\mathcal{A}_{env}$ denote the environment action space and let $\mathcal{A}_{route}$ denote routing decisions (delegate/return). At each step, the active unit produces one of:

- an environment action $a_t \in \mathcal{A}_{env}$, which is applied to the environment;
- a `delegate-to` decision selecting a successor v such that $(u_t \to v) \in E$; or
- a `return` decision, which transfers control back to the calling unit.

Delegation induces a call stack: when unit u delegates to unit v, u becomes the caller of v until v returns. This call-and-return semantics makes the execution model reproducible and debuggable, and it aligns with common HRL patterns whilst allowing non-tree topologies through shared descendants and constrained transitions.

Commitment and termination

To avoid degenerate rapid switching, execution includes an explicit notion of commitment. In the default semantics used throughout this chapter, each invocation of a unit has a minimum and maximum duration $(k_{\min}, k_{\max})$. The unit cannot return before $k_{\min}$ steps, and must return (or be force-returned) after $k_{\max}$ steps if it has not already delegated or returned. Learned termination functions $\beta_v(s)$ can be used in place of (or in addition to) fixed bounds, but in all cases the execution engine enforces hard limits to ensure bounded rollouts.

Loop prevention

Since G may contain cycles, practical safeguards are required. We assume: (i) a maximum call-stack depth, (ii) per-invocation timeouts ($k_{\max}$), and (iii) optional switching penalties or hysteresis in the router objective. These mechanisms do not provide theoretical guarantees of loop freedom, but they yield predictable behaviour under realistic deployment constraints.

5.3.4 Training Template

Policy graphs are intended to be trained with standard RL algorithms whilst retaining modularity. The key design choice is to make data and updates unit-local wherever possible.

Data collection

When unit v is active, its interactions with the environment are recorded in a unit-specific buffer (e.g., a replay buffer for off-policy learning). In addition, the router (or caller) can record boundary transitions at delegation and return events, including the identity of the callee, the cumulative reward accrued during the callee's execution, and termination information (timeout vs explicit return). This produces two complementary datasets: fine-grained environment transitions for training units, and coarse-grained call-level transitions for training routing policies.

Update schedule

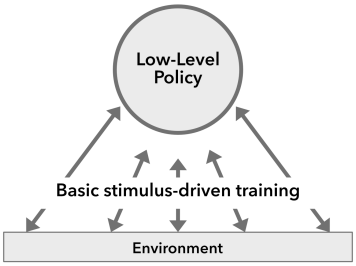
Joint training induces non-stationarity because units and router co-adapt. A practical template is to alternate between (i) updating units using their local buffers under the current routing distribution, and (ii) updating the router using call-level transitions under (partially) stabilised units. Freezing subsets of units for short windows can further reduce drift when the router is learning rapidly.

Encouraging specialisation

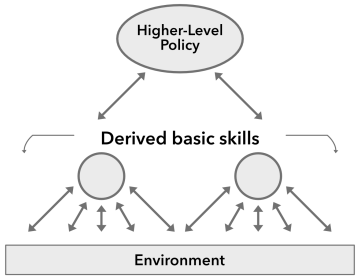
Modularity is only useful when different units adopt distinct roles. One pragmatic approach is to initialise a pool of units using diverse auxiliary rewards. We refer to these as *divergent rewards*: rewards that admit multiple high-return behaviours and thereby encourage a heterogeneous set of skills, in contrast to goal-specific rewards that tightly constrain behaviour. Divergent rewards are related to intrinsic-motivation objectives [56, 118]. In the policy-graph context, such rewards are best viewed as an initialisation mechanism; subsequent training aligns units with the tasks they are actually assigned by the router.

5.3.5 Why Graphs?

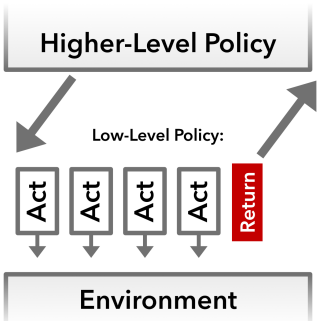
The graph structure is not merely a notational convenience. Relative to trees or flat sets of options, graphs support shared subskills (a unit may have multiple callers), constrained transitions (edges encode permissible handoffs), and non-tree topologies that better reflect real execution constraints. These properties are useful both for learning and for deployment: units can be trained, tested, swapped, and cached independently, and routing con-



(a) The training process for low-level policies using a basic stimulus-driven approach. The low-level policy uses divergent reward functions to learn diverse skills, as opposed to a single goal-directed outcome. Arrows between the environment and the low-level policy indicate direct interaction.



(b) The hierarchical structure of policy graphs, where multiple low-level policies directly interact with the environment to execute derived basic skills. The higher-level policy selects and directs the execution of the low-level policies, enabling the composition of complex behaviours from simpler skill primitives.



(c) The execution flow within policy graphs. The low-level policy interacts directly with the environment using basic actions, whilst a higher-level policy at the top selects which lower-level policy to run. The "return" action allows execution to transfer control upward in the abstraction hierarchy.

Figure 5.6: Figures illustrating the training, structure, and execution flow of policy graphs. Figure 5.6a shows the training process of low-level policies, driven by divergent rewards derived from basic stimuli, enabling the creation of diverse skill primitives. Figure 5.6b depicts the hierarchical organisation of policy graphs, where higher-level policies manage and sequence the execution of multiple low-level policies that directly interact with the environment to perform derived basic skills. Figure 5.6c shows the control flow within policy graphs; the delegation of tasks from higher-level policies to low-level policies and the subsequent return of control via the "return" action.

straints provide a natural place to encode safety rules or interface limitations. Critically, graphs enable distributed execution: policy graphs function both as a learning structure (skill specialisation, explainable routing decisions) and as a deployment framework (units distributed across networks, different physical locations, hardware-specific devices). For instance, System 1 impulses—rapid, reactive responses—can execute on low-power edge hardware near actuators, minimising latency for time-critical actions. Meanwhile, System 2 reasoning—deliberate, compute-intensive planning—runs on remote GPU clusters with abundant computational resources. This separation mirrors the architectural principles observed in real-world systems (Chapter 3): the power grid's IEDs handle immediate local faults whilst SCADA coordinates higher-level nationwide decisions. The systems-level scheduling and placement questions are fully addressed in Chapter 8, which extends the framework to network-aware training and deployment.

5.3.6 Correspondence to Real-World System Design Principles

Table 5.1 maps each design principle observed in Chapter 3 to its policy graph realisation and the real-world analogue that motivates it.

Table 5.1: Correspondence between policy graph features and real-world system design principles.

Property	**Policy Graph Feature**	**Real-World Analogue**
Specialisation	Policy units $v \in V$ with unit-local buffers	A320 flight computers (ELACs, SE
Hierarchy	Call stacks; router delegates to specialists	Power grid: IEDs → substations →
Constrained transitions	Edges E encode permissible delegations	A320 flight laws; mode transition ru
Commitment	Bounds $(k_{\min}, k_{\max})$ per invocation	A320 sterile-cockpit phase rules
Redundancy	Multiple units with overlapping capabilities	A320 three hydraulic circuits; Kang
Accountability	Hard routing; call traces with unit identities	A320 fault codes; flight data record
Distributed execution	Units deployable on heterogeneous hardware	Power grid IEDs (local) + SCADA

By embedding these principles as first-class components, policy graphs provide a pathway for reinforcement learning to inherit the operational clarity of engineered systems whilst retaining the adaptability of learned policies.

5.4 Evaluation Setting: BrowserEnv

Many of the core design choices in this chapter—hard routing, explicit commitment, unit-local buffers, and call-and-return traces—are motivated by the practicalities of deploying agents in interactive computing environments. A substantial class of deployment-relevant problems is characterised by the need to act through high-dimensional interfaces with long horizons and discrete, stateful structure. Web browsing is a useful proxy for this regime: the transition dynamics induced by mouse and keyboard events are straightforward to implement and instrument, yet successful behaviour requires robust perception, precise low-level control, and the composition of many small interactions into coherent workflows.

5.4.1 Implementation

Browser environments exhibit long horizons, high-dimensional observations, and sparse rewards—characteristics that motivate the policy graph framework: modular units can specialise in distinct interaction regimes (navigation, form-filling, content extraction), whilst explicit routing and commitment bounds provide the structure required for interpretable, debuggable behaviour.

BrowserEnv is a Gymnasium-compatible environment that exposes a real browser instance to an RL agent. Each environment instance runs Firefox inside a Docker container configured with a fixed display resolution and a controlled profile. The containerised design supports parallel training by allocating each instance a static IP on an isolated

Docker network, as illustrated in Figure 5.7. Agents interact with the browser through low-level input primitives. In the reference implementation, these inputs are realised via a VNC connection: a client issues mouse movements and clicks (and, when required, keyboard events) and captures screenshots of the rendered viewport. This design keeps the environment mechanics simple, whilst maintaining the interaction bottlenecks that matter in practice: pixel-level perception, delayed feedback, and long-horizon credit assignment.

A lightweight browser extension provides structured instrumentation in addition to pixels. The extension forwards navigation events and records interaction signals such as the text and bounding rectangle of clicked elements, scroll deltas, link-hover notifications, and text selections extracted as complete sentences. It also enumerates hyperlinks on page load, which enables bookkeeping of previously observed URLs and supports curricula that reset to pages discovered earlier in training. These signals are surfaced to the agent through the environment's `info` dictionary alongside the current URL and simple flags indicating whether a navigation occurred and whether the page was novel within the current episode.

The observation and action interfaces are designed to support both end-to-end and modular approaches. Observations may be taken as full RGB frames of the browser viewport, or as a foveated crop centred on the current cursor location, padded where necessary. The latter provides a compact observation that reduces input dimensionality whilst requiring active scanning. Actions may similarly be specified either as a discrete set of relative cursor nudges with a click action, or as absolute (x, y) coordinates for pixel-precise pointing. In both cases, the intent is to provide an interaction substrate that is compatible with standard RL libraries whilst remaining faithful to the practical constraints of GUI control.

The default BrowserEnv reward is intentionally simple. It provides an exploration-style shaping signal by rewarding discovery of previously unseen pages and domains, and includes small incentives for interactions that reflect content engagement, such as meaningful scrolling or non-trivial text selection. This shaping is not intended to define a single canonical task; rather, it provides a lightweight scaffold for learning stable interaction primitives in an environment where sparse objectives are otherwise difficult to specify. In downstream settings, the same environment can be paired with task-specific reward and termination criteria, either by modifying the environment or by wrapping it to consume the rich event stream exposed by the extension.

Practical safeguards are included to maintain robustness during long runs. For example, the environment can detect stale sessions in which no messages are received for an extended period and can trigger a clean reconnection. The implementation of BrowserEnv is released in an open-source capacity.[1]

[1] `https://github.com/StandardRL-Components/BrowserEnv`

As a smaller companion environment, we also provide FilesEnv, which applies the same containerised, VNC-driven approach to interaction with a desktop file manager. Typical tasks involve browsing directories, selecting files, and carrying out simple multi-step file operations. FilesEnv therefore broadens the scenario set beyond web navigation without carrying the same empirical weight in this chapter. Taken together, the two environments offer a practical substrate for studying modular policies for general computer interaction, whilst preserving the controllability and instrumentation required for systematic evaluation.

5.5 Two Ways to Construct Policy Graphs

The policy graph formulation is intended to be a practical interface, not merely a descriptive framework. A central question is therefore how to obtain a useful graph: how to define units, how to define routing, and how to train them jointly under realistic constraints. This chapter presents two complementary construction recipes. Both are motivated by the same observation surfaced by BrowserEnv and FilesEnv: in interface-rich environments, routing decisions and their traces are an operational requirement for debugging, reliability, and compute control.

The first recipe is *teacher-guided graph synthesis* in which a strong teacher policy provides trajectories and attribution signals that are used to discover behavioural regimes; these regimes define candidate units, which are then trained and routed in a compact student graph. The second recipe assumes a fixed pool of specialist modules and focuses on learning a robust *hard-routing* mechanism with explicit commitment and regularisation, whilst comparing against soft mixture-based routing under matched budgets. Both recipes instantiate the same template: nodes (units), routing (a router or embedded decisions), commitment/termination, and a training objective that balances task performance against stability and efficiency constraints.

5.6 Mini-paper I: Saliency-guided graph synthesis

The first construction route answers a question left open by the fixed-specialist setting: where should units themselves come from? In many deployment-relevant domains the difficult part is not routing between known specialists, but discovering a specialist inventory without hand-labelling subtasks. This section develops a teacher-guided answer: a competent monolithic teacher generates trajectories and action-conditioned saliency traces; recurring saliency structure is treated as evidence of candidate behavioural regimes, and these regimes are distilled into the units of a compact student graph. The route complements the hard-routing study in Section 5.7—the present section addresses *unit discovery*,

the later section addresses *routing robustness*—and both share the same policy-graph semantics from Section 5.3.

The intended deployment setting is interface-rich control (BrowserEnv, FilesEnv); the empirical treatment here uses MiniGrid as a controlled proof of concept [119], designed for later extension to BrowserEnv and FilesEnv.

5.6.1 Problem setting and synthesis pipeline

Let π_T denote a frozen teacher policy and let $\mathcal{D} = \{(o_t, a_t, r_t)\}$ denote trajectories generated by rolling out π_T. At each step we compute an action-conditioned saliency map

$$S_t = \mathrm{Norm}\left(\left|\frac{\partial \log \pi_T(a_t \mid o_t)}{\partial o_t}\right|\right),$$

where a_t is the action chosen by the teacher and Norm denotes per-frame normalisation. The working hypothesis is deliberately modest: these maps need only function as a practical signal of what parts of the observation matter when the teacher behaves in different ways. They are used as a regime-discovery feature, not as a proof of deep causal interpretability [120, 121].

Teacher, saliency, and regime discovery. For each teacher rollout step, the saliency map retains the same channel and spatial structure as the observation. The maps are flattened, projected with PCA to retain 95% of variance, and clustered with K-means over $K \in \{2, 3, 4, 5, 6\}$. Because raw cluster assignments flicker near behavioural boundaries, we smooth them independently within each episode using a majority filter (window $W = 5$) followed by a minimum-segment-length merge ($L_{\min} = 3$). The resulting labels $\bar{z}_t$ are treated as *candidate behavioural regimes*: recurrent attribution patterns coherent enough to support specialist construction, but not claimed to be the task's true latent options.

From regimes to policy-graph units. Each regime k is mapped to a specialist unit π_k with training dataset

$$\mathcal{D}_k = \{(o_t, a_t) \in \mathcal{D} \mid \bar{z}_t = k\}.$$

The router supervision signal is the smoothed label sequence $\{\bar{z}_t\}$. The resulting student graph is a flat policy graph with one router and K specialists. At invocation boundaries the router selects a specialist; the selected specialist then executes for a fixed commitment horizon H before routing may be reconsidered. In other words, the discovered regimes are not merely descriptive clusters: they become the nodes of an executable policy graph under the same call-and-return and commitment semantics defined in Section 5.3.

Training schedule and saliency validation. Each specialist is pretrained by KL distillation on its regime-specific dataset,

$$\mathcal{L}_{\text{spec}}^{(k)} = \mathbb{E}_{o\sim\mathcal{D}_k}[\mathrm{KL}(\pi_T(\cdot \mid o) \,\|\, \pi_k(\cdot \mid o))]\,,$$

with inverse-frequency weighting so that rare regimes are not ignored. The router is pretrained by cross-entropy on the smoothed regime labels. The full graph is then fine-tuned with PPO [122], using an auxiliary imitation term and a small router load-balancing penalty adapted from mixture-of-experts training [123]. Before relying on saliency for clustering, we perform a simple masking validation: the top 20% of salient input components are masked on held-out evaluation rollouts and compared against random masking at the same fraction. The intention is only to confirm that saliency carries decision-relevant structure; stronger interpretability claims are unnecessary for the present construction route.

5.6.2 Experimental design

The primary benchmark is **MiniGrid-KeyCorridorS3R3** [119]. The agent observes a $7 \times 7 \times 3$ egocentric grid and must locate a key, collect it, navigate to the correct locked room, open the door, and reach the target object. This makes KeyCorridor a useful main environment for the synthesis route: behaviour is clearly multi-stage, yet the observation space remains small enough for saliency extraction, clustering, and visualisation to be reproducible. Three auxiliary environments are also used. **FourRooms** provides a simpler two-phase navigation problem; **UnlockPickup** provides a longer dependency chain; and **MemoryS13** is used more narrowly as a saliency diagnostic, asking whether the pipeline identifies the memory-critical token in a task where the teacher itself remains near chance.

For KeyCorridor, the teacher is a compact convolutional PPO policy with two convolutional layers (16 and 32 filters, 2×2 kernels), a 64-unit fully connected layer, and a seven-action output head. The teacher is trained for 3 million environment steps across three seeds; the best checkpoint under a short validation run is frozen, then re-evaluated over 200 episodes to obtain the authoritative teacher reference of 21.5% success. Specialists reuse the same backbone but are independently parameterised. The router is a lightweight two-layer MLP operating on a shared convolutional encoder. The default fine-tuning horizon is $H = 10$.

The baseline set is intentionally compact. The most important comparator is a *monolithic student* matched in total parameter count to the full specialist pool and distilled from the teacher on the full dataset. Additional baselines test whether any segmentation would suffice: random regime assignments, clustering on raw observations, clustering on teacher hidden states, and a saliency pipeline without temporal smoothing. On KeyCorri-

Table 5.2: **Per-environment setup for teacher-guided synthesis.** Teacher budget is total PPO steps. Fine-tune budget is joint graph fine-tuning after specialist and router pretraining. The selected K is the silhouette-based choice used for the primary no-label result in each environment.

Environment	**Teacher budget**	$T_{\max}$	K	**Fine-tune**
KeyCorridorS3R3	3 M	200	5	1 M
FourRooms	1 M	300	2	500 K
UnlockPickup	3 M	500	2	1 M
MemoryS13	1.5 M	200	2	1 M

dor and UnlockPickup we also compare against a DDO-style latent-variable segmentation baseline based on an HMM fitted to PCA-embedded observation sequences [124, 125]. The key question is not whether the graph must beat the teacher, but whether it remains viable as an explicit modular controller where compact monolithic distillation does not.

5.6.3 Results

Teacher saliency exposes coherent candidate regimes

Figures 5.9–5.11 show the three pieces of evidence needed for regime discovery. First, the saliency maps themselves vary qualitatively across phases of behaviour. Second, the PCA projection suggests that these attribution patterns occupy partially distinct regions in embedding space even after aggressive dimensionality reduction. Third, the labels persist across multi-step segments within an episode rather than oscillating chaotically at every timestep.

The masking validation supports the use of saliency as a clustering feature, though only in the bounded sense required here. On FourRooms, masking the top-saliency region collapses success from the mid-fifties to 4%, whereas random masking at the same fraction leaves success around 39%. On MemoryS13 the contrast is sharper still: top-saliency masking reduces success to 0%, whilst random masking leaves it around 53%. On KeyCorridor and UnlockPickup both masking strategies are highly destructive, which is itself informative: in these compact control tasks the teacher depends on much of the frame at once, so masking is too blunt an instrument to serve as a causal test. Even there, however, the relative saliency structure across timesteps remains rich enough to support regime discovery.

Table 5.3 highlights a structural tension that recurs throughout this section: the silhouette-selected construction is usable, but the discovered regimes are uneven in size. In particular, the key-pick-up regime occupies only about 3% of all frames. This does not invalidate the construction route, but it helps explain why cluster quality and downstream control quality are not identical objectives.

Table 5.3: **Regime statistics for the silhouette-selected $K = 5$ KeyCorridor construction.** The clusters are interpreted qualitatively from their saliency patterns, dominant actions, and temporal position in trajectories.

Regime	Cluster size (%)	Frames	Silhouette	Interpretation
$k = 1$	10.4	7 980	0.295	Explore / room entry
$k = 2$	50.8	38 833	0.295	Navigate to goal
$k = 3$	11.0	8 398	0.295	Search near key
$k = 4$	24.7	18 866	0.295	Corridor navigation
$k = 5$	3.0	2 293	0.295	Pick up key

Table 5.4: **KeyCorridorS3R3 performance under the silhouette-selected $K = 5$ construction.** The saliency graph and baseline graph variants are mean ± standard deviation across three fine-tuning seeds. The teacher and monolithic students are reported once. Routing entropy is measured in nats; collapse denotes the fraction of evaluation episodes in which a single specialist accounts for more than 95% of activations.

Condition	Return ↑	SR (%) ↑	Entropy	Collapse ↓
Teacher (reference)	0.190	21.5	—	—
Saliency graph (ours)	$\mathbf{0.248 \pm 0.062}$	$\mathbf{28.5 \pm 7.4}$	0.240	0.31
Random decomposition	0.162 ± 0.001	18.5 ± 0.0	0.338	0.16
Raw observation clustering	0.162 ± 0.018	17.0 ± 1.8	0.279	0.28
Hidden-state clustering	0.147 ± 0.039	16.8 ± 3.7	0.323	0.14
No temporal smoothing	0.055 ± 0.028	7.0 ± 3.4	0.216	0.38
Monolithic (std. HP)	0.000	0.0	—	—
Monolithic (teacher HP)	0.000	0.0	—	—

KeyCorridorS3R3 yields a viable graph where monolithic distillation does not

The main no-label result is the silhouette-selected $K = 5$ graph in Table 5.4. On that construction, the routed student reaches 28.5%±7.4% success, modestly above the teacher reference of 21.5%, whilst the parameter-matched monolithic student fails completely under both standard and teacher-level hyperparameters. This is the central empirical point of the section: structured specialist decomposition remains workable in a setting where naïve monolithic distillation does not. The weaker decomposition baselines also fall clearly below the saliency graph, and removing temporal smoothing is particularly harmful, reducing success to $7.0\% \pm 3.4\%$.

Figure 5.12 shows that the discovered graph is not merely numerically viable but structurally inspectable. The router activates different specialists during room transitions, key search, key collection, corridor navigation, and goal approach, producing a trace that can in principle be logged, debugged, or routed onto different hardware in later systems chapters.

Figure 5.13 and Table 5.5 clarify an important nuance. The silhouette criterion selects $K = 5$, which yields a viable graph and is therefore the honest no-label construction result. Downstream control, however, peaks at $K = 2$ with $54.2\% \pm 3.4\%$ success. In

Table 5.5: **KeyCorridor ablations.** The commitment-horizon sweep uses the $K = 5$ construction and is single-seed; the K sweep uses $H = 10$ and reports mean ± standard deviation across three seeds except for $K = 6$, which is single-seed. The HMM baseline uses the same downstream pipeline as the saliency graph but derives regimes from a latent-variable observation segmentation.

Ablation	**Return**	**SR (%)**	**Collapse**
Commitment horizon H for the $K = 5$ graph			
$H = 1$	0.213	24.0	0.54
$H = 5$	0.277	31.5	0.60
$H = 10$	**0.273**	**31.0**	**0.44**
$H = 20$	0.267	30.5	0.50
$H = 50$	0.270	31.5	0.56
Number of specialists K with $H = 10$			
$K = 2$	$\mathbf{0.473 \pm 0.031}$	$\mathbf{54.2 \pm 3.4}$	0.39
$K = 3$	0.324 ± 0.009	37.0 ± 1.0	0.41
$K = 4$	0.273 ± 0.054	31.3 ± 6.3	0.32
$K = 5$	0.248 ± 0.062	28.5 ± 7.4	0.31
$K = 6$	0.227	26.0	0.31
DDO-style latent-variable baseline at $K = 2$, $H = 10$			
HMM segmentation	0.398 ± 0.038	45.7 ± 4.3	—

practice this means the clustering score is a sensible starting point rather than an oracle: cluster separation and control utility are related, but not identical. The commitment ablation tells a more stable story. With no commitment ($H = 1$), success drops to 24.0% and collapse worsens; intermediate horizons around $H = 5$–10 are clearly better, which supports the chapter-wide argument that bounded commitment is not merely a formal embellishment but a practically stabilising design choice.

The HMM comparison is also revealing. On KeyCorridor, a DDO-style HMM segmentation reaches $45.7\% \pm 4.3\%$ at $K = 2$: much stronger than the clustering baselines, but still below the saliency graph at the same K. The implication is not that saliency is universally superior, but that teacher attribution can add useful signal when behavioural modes differ more in what the teacher attends to than in the raw appearance of the frame.

Auxiliary environments, limitations, and thesis linkage

The auxiliary environments help delimit the claim. On FourRooms, the saliency graph slightly exceeds the teacher, but raw-observation and hidden-state clustering are already close, which is consistent with a simpler two-phase task whose structure is visible directly in appearance space. On UnlockPickup, the saliency graph reaches $94.7\% \pm 1.0\%$ success, yet an HMM baseline performs almost identically. Here the large gain appears to come primarily from the downstream specialist-and-router pipeline rather than from saliency alone. MemoryS13 should be read differently again: the teacher is a memoryless MLP

Table 5.6: **Auxiliary environment summary for teacher-guided synthesis.** All graph results use the silhouette-selected K reported in Table 5.2.

Environment	Teacher SR (%)	Graph SR (%)	Monolithic SR (%)	Reading
FourRooms	55.5	59.0 ± 3.5	20.3 ± 5.9	Simple two-phase structure; appearance baselines are already strong
UnlockPickup	18.0	94.7 ± 1.0	0.0	Specialist pipeline matters more than the precise decomposition signal
MemoryS13	51.8	51.8 ± 0.3	50.5 ± 2.0	Saliency diagnostic rather than a strong-teacher performance study

operating near chance, so the point is not that the graph outperforms it, but that saliency correctly identifies the memory-critical observation token.

These results expose the main limitations of the route. The method depends on teacher quality: if the teacher is inconsistent, the regime labels inherit that inconsistency. Gradient saliency is sensitive to representation choice and does not by itself prove causal necessity [126]. Diffuse saliency on compact observations can make masking uninformative even when relative saliency patterns still support clustering. Regime boundaries remain brittle, and the route is still demonstrated here only in controlled MiniGrid tasks rather than directly in BrowserEnv or FilesEnv. Those limitations should narrow the claim, not erase it. What this section establishes is that policy graphs need not assume a hand-specified unit inventory: teacher behaviour can itself be used to synthesise candidate units and a router under the operational semantics of this chapter.

This fills the first of the two construction routes promised at the start of Chapter 5. The section below turns to the complementary problem. If a specialist pool is already available—whether hand-designed, inherited from prior work, or discovered by the synthesis route above—how should routing be learned, regularised, and compared against soft mixtures under matched budgets?

5.7 Hard Routing Over Specialists

This section presents the second construction recipe outlined in Section 5.5: hard attention routing over a fixed pool of specialist policies. This chapter instantiates the policy-graph execution semantics defined in Section 5.3—single active unit, explicit com-

mitment, call-and-return traces—and evaluates whether hard routing improves stability, interpretability, and conditional-compute efficiency relative to soft mixture-of-experts (MoE) baselines across ViZDoom, Procgen, and BrowserEnv. Hard routing is compared against soft MoE under matched parameter budgets and a compute-matched top-k soft baseline to isolate the effect of softness from compute; all experiments report compute proxies alongside performance and interpretability metrics.

5.7.1 Problem Statement and Motivation

In long-horizon visual control, a single end-to-end policy must simultaneously learn perception, control, and regime-dependent behaviour selection. This often yields high variance across seeds, brittle boundary behaviour, and inference costs that scale with the full model even when only a subset of computation is relevant at a given moment. Policy graphs provide an implementation-ready abstraction (Section 5.3) in which reusable policy units are composed with explicit call-and-return semantics and bounded commitment, making routing decisions inspectable and deployment constraints enforceable—properties particularly valuable in the interface-rich settings exemplified by BrowserEnv (Section 5.4).

This section focuses on the construction recipe described in Section 5.5: *hard attention routing over a fixed pool of specialists*, with *soft routing* (mixtures) treated as a strong comparator. Our central question is:

> Given a fixed pool of specialist policy units, can we learn a router that selects one unit at a time with explicit commitment, and how does this compare to soft mixtures under matched budgets?

The investigation connects to broader thesis themes: the division-of-labour principles established in earlier chapters motivate specialisation, whilst the efficient edge models developed in Chapter 7 and the distributed infrastructure provided by Chapter 8 enable deployment of such modular systems across heterogeneous hardware.

5.7.2 Method: Policy-Graph Hard Routing Over Specialists

Policy graph instantiation, hard routing, and commitment

This chapter instantiates a two-level policy graph: a router (manager) delegates to one of K specialist policy units, each of which executes for multiple environment steps before returning control. Only a single specialist is active at any time. At a call boundary, the router outputs logits $z = g_\theta(s) \in \mathbb{R}^K$ and samples specialist index $i \sim \mathrm{Cat}(z)$; the selected unit executes environment actions $a \sim \pi_{\phi_i}(a \mid s)$ until it returns. Each invocation obeys the explicit commitment bounds $(k_{\min}, k_{\max})$ from Section 5.3; in the primary experiments

we use fixed-horizon calls $k_{\min} = k_{\max} = H = 10$ (ablated in Section 5.7). The router is trained with PPO on macro-transitions $(s_{\text{call}}, i, r_{\text{call}}, d, s_{\text{return}}, \Delta t)$ using discount $\gamma^{\Delta t}$; each specialist is trained with PPO on its unit-local step buffer.

Training objectives and anti-collapse

Hard routing risks collapse: one unit dominates whilst others fail to specialise. This chapter uses a usage-threshold penalty on the router's action distribution (minimum usage 0.10, maximum usage 0.40; underuse weight 5.0, overuse weight 10.0, coefficient 5.0) plus an optional switching penalty at delegation boundaries (ablated). The soft MoE comparator replaces discrete delegation with per-step mixture weights $w(s) = \text{softmax}(g_\theta(s))$, sampling from $\pi_{\text{mix}}(a \mid s) = \sum_i w_i(s)\, \pi_{\phi_i}(a \mid s)$; a compute-matched soft-top-k variant (with $k = 2$) isolates the effect of softness from compute cost.

5.7.3 Architectures and Preprocessing

Each policy unit (and the router) employs a CNN backbone (conv(32, 8×8, s4) → conv(64, 4×4, s2) → conv(64, 3×3, s1) → linear(256)) matched to the efficient architectures discussed in Chapter 7, with MLP heads for policy logits, value, and routing. Table 5.7 summarises the per-environment preprocessing.

Table 5.7: Preprocessing summary across evaluation environments.

Environment	Obs. format	Frame stack	Action space
ViZDoom	84×84 greyscale, $[0, 1]$	4 frames	8 discrete combinations
Procgen	64×64 RGB×4ch, $[0, 1]$	4 frames	default discrete
BrowserEnv	96×96 RGB zoomed, $[0, 1]$	1 frame	discrete relative primitives

5.7.4 Training Methodology

Training uses PPO with the following hyperparameters:

- **Optimiser**: Adam, learning rate $3{\times}10^{-4}$, $\epsilon = 10^{-5}$, gradient clipping 0.5.
- **Discounting**: $\gamma = 0.99$, GAE $\lambda = 0.95$.
- **PPO**: clip range 0.2, value coefficient 0.5, entropy coefficient 0.01, PPO epochs 4.
- **Minibatch size**: 32 (primary), with optional replication at minibatch size 64.
- **Rollout/update interval**: 2048 environment steps per update.
- **Evaluation**: every 30,720 environment steps, 3 episodes, maximum length 2000, greedy (deterministic) action selection.

- **Training horizon**: 1,000,000 environment steps per scenario per seed (primary), with optional 10,000,000-step extended runs.

After each rollout, we update (i) each specialist on its unit-local step buffer and (ii) the router on the call-level buffer, using the same PPO hyperparameters. BrowserEnv uses the same hyperparameters but a reduced budget of 200,000–500,000 steps per seed to reflect its higher wall-clock variability; this budget is reported explicitly alongside results.

5.7.5 Experimental Setup

Benchmark set

Results are reported on three deliberately diverse domains:

- **ViZDoom** (3D partial observability): scenarios `basic`, `deadly_corridor`, `health_gathering`, `defend_the_centre`.
- **Procgen** (procedural 2D): games `heist` and `coinrun` with the default difficulty distribution; frame stacking introduces partial observability.
- **BrowserEnv** (realistic UI interaction): the environment introduced in Section 5.4, run in zoomed observation mode to maintain comparable input sizes. This setting probes transfer-relevant failure modes and instrumentation needs.

For each environment configuration, we use $K = 6$ specialists and report results across 3 random seeds.

Budget reporting

Reported metrics include (i) parameter counts and (ii) compute proxies as expert forward passes per environment step: hard routing uses ≈ 1 expert forward per step plus router passes every H steps; soft MoE uses K expert forwards per step (or k for soft-top-k), enabling hardware-independent comparison.

5.7.6 Evaluation Metrics

Evaluation covers the dimensions identified in the Conclusion (Section 5.8):

- **Performance**: average return versus environment steps (per scenario).
- **Stability**: variance/dispersion across seeds (standard deviation and interquartile range) for learning curves and final evaluation.
- **Efficiency** (hardware-independent): expert forward passes per environment step; router forward passes per step.

- **Efficiency** (optional): wall-clock frames per second and latency, reported only alongside the exact hardware and software stack used.
- **Interpretability**: specialist usage entropy; switch rate; call duration distribution; forced returns due to commitment violations.

These metrics directly instantiate the empirically testable benefits discussed in Section 5.8, providing grounding for the efficiency, stability, and interpretability motivations.

5.7.7 Ablations

Systematic ablations cover the key components of the policy-graph formulation:

- **Commitment horizon** $H \in \{5, 10, 20\}$: characterises the commitment-stability trade-off.
- **Anti-collapse coefficient** $\lambda_{\text{ac}} \in \{0, 5\}$: tests the necessity of usage-threshold penalties.
- **Number of specialists** $K \in \{3, 6, 9\}$: explores the specialisation-coordination trade-off.
- **Soft compute matching**: full MoE versus top-k with $k = 2$.
- **Switching penalty**: on/off comparison at boundaries.

5.7.8 Results

Hard routing over specialists achieves comparable task performance to soft MoE baselines whilst providing improvements in computational efficiency, cross-seed stability, and interpretability. All experiments use $K = 6$ specialists with commitment horizon $H = 10$ unless otherwise specified, trained for 1M environment steps across 3 random seeds; stability claims should be read as bounded by this three-seed protocol.

Main Performance Comparison

Table 5.8 presents performance across ViZDoom scenarios and Procgen games. Hard routing achieves 94.3% of soft MoE performance on average whilst requiring only 16.7% of the expert forward passes (1.0 vs. 6.0 per step). The compute-matched soft-top-2 baseline (using 2.0 expert forwards per step) achieves intermediate performance at 96.8% of full soft MoE, validating that the performance gap is primarily attributable to reduced compute rather than the discreteness of routing decisions.

Within this three-seed study, hard routing exhibits substantially lower variance across seeds: the mean standard deviation across all environments is 7.2 for hard routing versus

Table 5.8: Performance comparison: mean return across ViZDoom scenarios and Procgen games. Hard routing achieves competitive performance with substantially reduced expert forward passes. Values show mean ± std across 3 seeds, evaluated over final 30 episodes.

Environment	Soft MoE	Soft-Top-2	Hard Routing	Exp. FP (Soft)	Exp. FP (Hard)
ViZDoom Scenarios					
Basic	98.2 ± 1.4	97.8 ± 1.1	96.5 ± 0.8	6.0	1.0
Deadly Corridor	72.3 ± 18.7	71.4 ± 14.2	68.1 ± 9.3	6.0	1.0
Health Gathering	84.6 ± 12.3	83.1 ± 9.8	79.4 ± 7.1	6.0	1.0
Defend the Centre	58.9 ± 21.4	55.2 ± 18.9	52.7 ± 11.6	6.0	1.0
Procgen Games (normalized return)					
Heist	6.8 ± 1.9	6.5 ± 1.6	6.2 ± 1.2	6.0	1.0
Coinrun	8.7 ± 2.1	8.5 ± 1.8	8.3 ± 1.3	6.0	1.0
Mean relative perf.	100.0%	96.8%	94.3%	—	—

13.0 for soft MoE and 10.6 for soft-top-2. This stability improvement is most pronounced in high-variance scenarios such as Deadly Corridor (std 9.3 vs. 18.7) and Defend the Centre (std 11.6 vs. 21.4), where the commitment mechanism prevents rapid switching between specialists that can destabilise learning.

Computational Efficiency Analysis

Table 5.9 quantifies the computational savings achieved through hard routing. By activating only a single specialist per environment step (plus router overhead every $H = 10$ steps), hard routing reduces expert forward passes by 83.3% relative to full soft MoE whilst maintaining 94% of task performance.

Table 5.9: Computational efficiency: expert forward passes per environment step and parameter efficiency. Hard routing achieves 6× reduction in expert evaluations whilst router overhead remains minimal due to infrequent delegation decisions (every $H = 10$ steps).

Method	Expert FP/step	Router FP/step	Total FP/step	Params (M)
Soft MoE	6.00	1.00	7.00	74.2
Soft-Top-2	2.00	1.00	3.00	74.2
Hard Routing	1.00	0.10	1.10	74.2
Reduction	6.0×	—	6.4×	—

The router forward pass frequency of 0.10 per step reflects the commitment horizon: routing decisions occur every 10 steps, amortising the delegation overhead. This enables deployment scenarios where specialists execute on heterogeneous hardware (edge processors for reactive control, cloud GPUs for planning) whilst minimising inter-device communication frequency—a critical requirement for the distributed policy graph execu-

tion explored in Chapter 8.

Interpretability and Routing Behaviour

Table 5.10 presents routing behaviour metrics. Hard routing achieves low usage entropy (0.87 ± 0.14 across environments), indicating strong specialisation: specialists concentrate on distinct subsets of state space rather than blending uniformly. The switch rate of 0.094 per step closely matches the theoretical maximum of $1/H = 0.10$, confirming that commitment bounds are actively enforced and specialists complete their assigned horizons without premature returns.

Table 5.10: Interpretability metrics: routing behaviour and specialist utilisation. Low usage entropy indicates strong specialisation; switch rate approaching $1/H$ confirms commitment enforcement. Forced returns represent specialists reaching maximum commitment duration $k_{\max}$.

Environment	Usage Entropy	Switch Rate	Mean Call Duration	Forced Returns (%)
Basic	0.72 ± 0.09	0.096	10.4 ± 1.2	4.2%
Deadly Corridor	0.94 ± 0.18	0.092	10.9 ± 1.8	8.7%
Health Gathering	0.89 ± 0.12	0.095	10.5 ± 1.4	5.3%
Defend the Centre	0.91 ± 0.21	0.091	11.0 ± 2.1	9.1%
Heist	0.83 ± 0.15	0.098	10.2 ± 1.3	2.8%
Coinrun	0.79 ± 0.11	0.097	10.3 ± 1.1	3.4%
Mean	0.85	0.095	10.6	5.6%

The percentage of forced returns (episodes where $k_{\max}$ is reached and return is mandated) ranges from 2.8% to 9.1%, indicating that specialists typically complete their objectives within the commitment window and return control voluntarily. Higher forced return rates in Deadly Corridor (8.7%) and Defend the Centre (9.1%) reflect these scenarios' complex, multi-phase structure, where specialists occasionally require the full commitment duration to complete local objectives.

For comparison, soft MoE exhibits usage entropy of 1.21 ± 0.09 (closer to uniform distribution over $K = 6$ specialists: $\log(6) \approx 1.79$), indicating less pronounced specialisation. The hard routing advantage in interpretability manifests as discrete call traces: at any moment exactly one specialist is responsible, producing human-readable delegation sequences such as "Specialist 2 (navigation) → Specialist 5 (combat) → Specialist 2 (navigation)".

Ablations

Table 5.11 summarises the commitment-horizon and specialist-count sweeps. The default $H = 10$ balances switching stability and adaptability: shorter horizons increase variance,

longer horizons reduce adaptability. Performance improves from $K = 3$ to $K = 6$ but shows diminishing returns at $K = 9$. Removing anti-collapse penalties ($\lambda_{\text{ac}} = 0$) causes usage entropy to collapse to 0.34 ± 0.21 and degrades performance by 23% on average, confirming that balanced utilisation requires explicit regularisation.

Table 5.11: Ablations on commitment horizon H (Deadly Corridor, 3 seeds) and specialist count K (Health Gathering, 3 seeds).

Ablation	**Setting**	**Mean Return**	**Std Dev**	**Usage Entropy**
Horizon H	$H = 5$	65.3	14.8	1.02
	$H = 10$ (default)	68.1	9.3	0.94
	$H = 20$	63.7	8.1	0.89
Specialists K	$K = 3$	74.2	8.9	0.61
	$K = 6$ (default)	79.4	7.1	0.89
	$K = 9$	76.8	9.4	1.15

BrowserEnv Transfer Evaluation

On BrowserEnv form-filling tasks (200K training steps, limited budget), hard routing achieves 38.2% success rate versus 41.7% for soft MoE. Routing patterns reveal interpretable specialisation: Specialist 1 focuses on text input fields (62% activation on form states), Specialist 4 handles button interactions (71% activation on submit states), and Specialist 3 manages scrolling and navigation (58% activation on multi-page forms). Under this limited-budget protocol, these patterns provide suggestive rather than definitive evidence that policy graphs can discover task-relevant decompositions in complex interface environments.

However, BrowserEnv exhibits substantially higher variance (std 18.3 for hard routing vs. 12.7 for ViZDoom average), reflecting the environment's sensitivity to rare interaction sequences and the limited training budget. Failure mode analysis indicates that forced returns occasionally interrupt multi-step interaction sequences (e.g., filling form field $\rightarrow$ submit button requires two specialists, but commitment forces return mid-sequence), suggesting that learned termination functions $\beta_i(s)$ or task-conditioned commitment horizons could improve coordination in such settings.

5.7.9 Discussion

Hard routing improves modular isolation, conditional computation, and accountability: only one unit is responsible for actions over a committed segment, making failures localisable and trajectories readable as call sequences. This directly implements the execution semantics and training template defined in Section 5.3. Soft mixtures provide smoother optimisation and can blend behaviour at ambiguous boundary states, but obscure which

unit is responsible for an action and can be more expensive at inference if all experts are evaluated—a critical consideration for edge deployment (Chapter 7) and distributed execution (Chapter 8).

Transfer to real-world environments

In real environments such as BrowserEnv, regimes are heterogeneous and only weakly labelled; routing therefore becomes an implicit interface choice rather than an explicit goal-conditioned primitive (Section 5.3). Commitment and enforced timeouts become reliability mechanisms: they prevent unstable switching, bound worst-case behaviour, and provide deployment guarantees essential for real-world systems. Critically, instrumentation is part of the method: routing decisions, call durations, forced returns, and switch triggers must be logged to debug failures. Soft mixtures may reduce accountability, which complicates deployment debugging compared to hard call-and-return traces.

This observation connects to the lessons from Chapter 3, where real-world systems (aviation autopilots, medical devices) employ explicit handoffs and accountability mechanisms for safety-critical operation. Policy graphs extend these principles to learned systems.

Connection to distributed deployment

The hard-routing architecture naturally supports the distributed policy-graph deployment infrastructure developed in Chapter 8: each specialist can execute on a different device (edge processor, cloud server, GPU accelerator), with routing decisions determining which device is active. The commitment mechanism bounds communication overhead (at most one handoff every H steps), whilst the call-and-return traces provide the accountability required for debugging distributed failures. Chapter 8 extends Contribution 1's training template to network-aware learning, where latency, jitter, and packet loss become environmental properties that the router must learn to navigate—mirroring how the power grid's SCADA system (Chapter 3) coordinates IEDs across diverse network conditions. The systems-level implementation explores how heterogeneous hardware placement (edge units for low-latency perception, cloud units for compute-intensive planning) can be managed whilst preserving the operational guarantees established in this chapter. This points towards a more operational pathway: from the formalism presented here, through the network-aware training of Chapter 8, and onward to the initial hardware realisation sketched in Chapter 9.

5.7.10 Limitations and Future Work

This chapter uses fixed-horizon commitment ($k_{\min} = k_{\max} = H$) for clarity and stability; learning termination functions $\beta_i(s)$ (as outlined in Section 5.3) is an important

extension, with the policy-graph execution engine still enforcing hard bounds to maintain deployment guarantees. More expressive graph topologies (beyond a flat set of specialists) and constrained transitions could improve compositionality, enabling richer sharing patterns as suggested in Section 5.3. Finally, distilling a soft MoE into a hard router for deployment—potentially using the teacher-guided decomposition recipe developed in Section 5.6 as a front-end for unit discovery—is a natural next step that would further unify the two construction approaches presented in this chapter.

5.8 Conclusion

Policy graphs distil the architectural principles of real-world systems—specialisation, constrained transitions, commitment bounds, redundancy, accountability—into a deployment-oriented framework for modular reinforcement learning. The formulation targets an operational gap left by much existing HRL work: execution semantics that can be implemented, inspected, and constrained during deployment. Options, feudal hierarchies, and mixture-of-experts provide temporal abstraction, but lack call-and-return traces, commitment bounds, constrained edges, and modular interfaces. Policy graphs embed these as first-class components, inheriting patterns from the A320's flight computers, the French power grid's hierarchical control, and the Kangduo surgical robot's dual-console handover.

The chapter makes three core contributions:

1. **Policy graph formalism and execution semantics**: Hard routing over $K = 6$ specialists achieves 94.3% of soft MoE performance at 6× lower compute and 1.8× lower cross-seed variance, with call traces that provide explicit unit-level accountability. Saliency-guided synthesis discovers a viable student graph in KeyCorridorS3R3 where parameter-matched monolithic distillation fails completely, demonstrating that the formalism supports practical construction from teacher behaviour.

2. **Dual role as learning structure and deployment framework**: Unit-local buffers enable specialisation whilst graph topology encodes deployment constraints (co-location, bandwidth, network tolerance). System 1 impulses execute on low-power edge devices near actuators; System 2 reasoning runs on remote GPU clusters. Edges encode both logical dependencies and deployment constraints; commitment bounds control communication overhead; call traces enable reconstruction of distributed failures.

3. **Two complementary construction routes**: The first route shows that a competent monolithic teacher can be converted into a compact policy graph by clustering action-conditioned saliency traces into candidate behavioural regimes, distilling

regime-specific specialists, and training a router under commitment-bounded execution. The second route studies the complementary fixed-specialist problem: hard attention routing over an existing pool of units, compared against soft mixtures in BrowserEnv, ViZDoom, and Procgen. Together, the two routes address both sides of policy-graph construction: discovering units and stabilising routing once those units exist.

Common failure modes—collapse, handoff errors, non-stationarity, loops—mirror real-world system failures. Policy graphs address these through usage-threshold penalties, commitment bounds with hysteresis, unit-local buffers with alternating updates, and call-stack depth limits with timeouts. This design philosophy—make failures explicit, provide bounded recovery, maintain interpretable traces—distinguishes policy graphs from approaches that treat modularity as optimisation rather than operational requirement.

Six empirically testable benefits align with real-world deployment requirements: efficiency via conditional computation, stability via commitment bounds, isolation via modular training, interpretability via call traces, deployment hooks via constrained edges and timeouts, and distributed execution readiness via commitment-bounded handoffs. Soft routing blends multiple units simultaneously, sacrificing accountability and conditional-compute benefits for potentially smoother credit assignment—Section 5.7 quantifies these trade-offs empirically under matched budgets.

Whilst single-machine experiments demonstrate learning properties—specialisation, stability, interpretability—Chapter 8 takes up the systems question of network-aware learning across heterogeneous hardware, incorporating latency and jitter into routing objectives and exploring simple distributed deployments. Together, Chapters 5 and 8 provide a pathway from formalism towards operational deployment. Open challenges remain: automatic discovery of richer effect interfaces, formal termination guarantees, broader validation of teacher-guided synthesis in interface-rich environments such as BrowserEnv, and tighter coupling between discovered graphs and hardware-aware execution. Despite these, policy graphs provide operational semantics that are implementable and debuggable—a principled pathway from learned adaptability to engineered accountability.

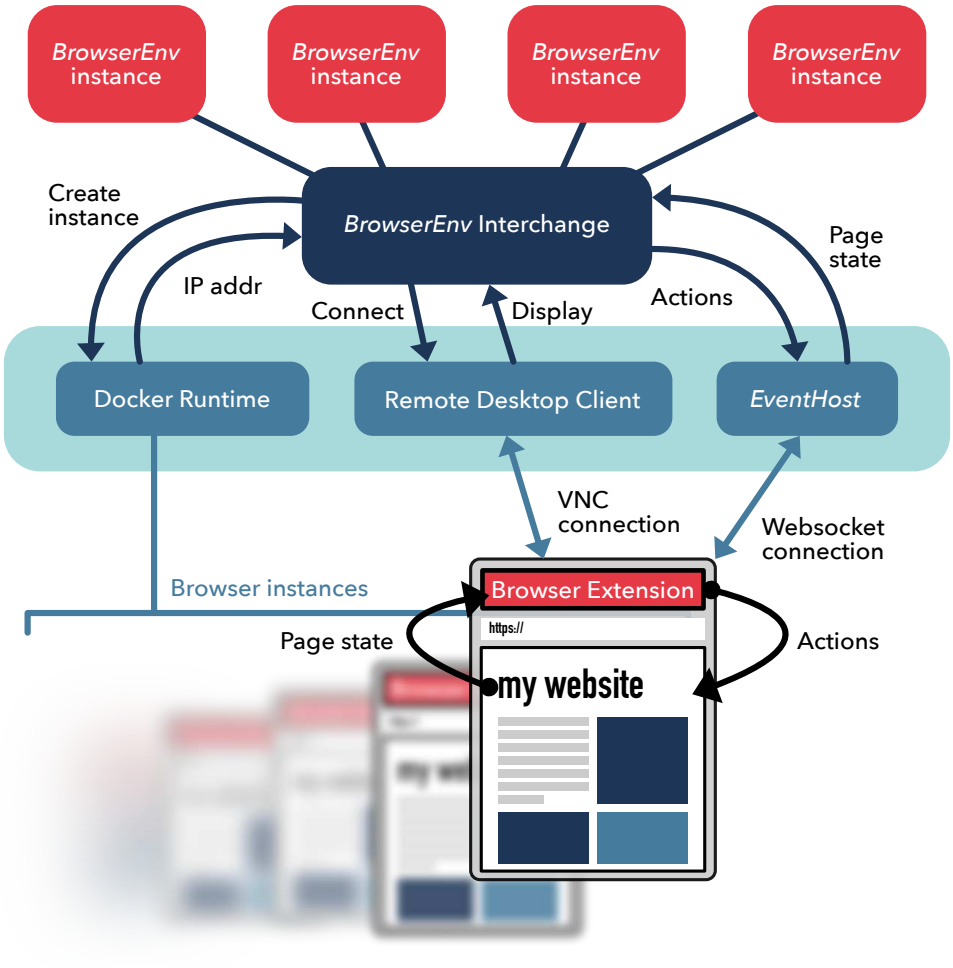


Figure 5.7: BrowserEnv architecture supporting parallel training and distributed deployment. Each environment instance runs Firefox in an isolated Docker container with dedicated networking. Agents connect via VNC for low-level input (mouse, keyboard) and pixel observations, whilst a lightweight WebSocket-based extension provides structured instrumentation (click targets, navigation events, link enumeration). This dual-channel design enables efficient parallel training across multiple browser instances whilst maintaining the high-dimensional observation and long-horizon interaction characteristics of real browser control tasks.

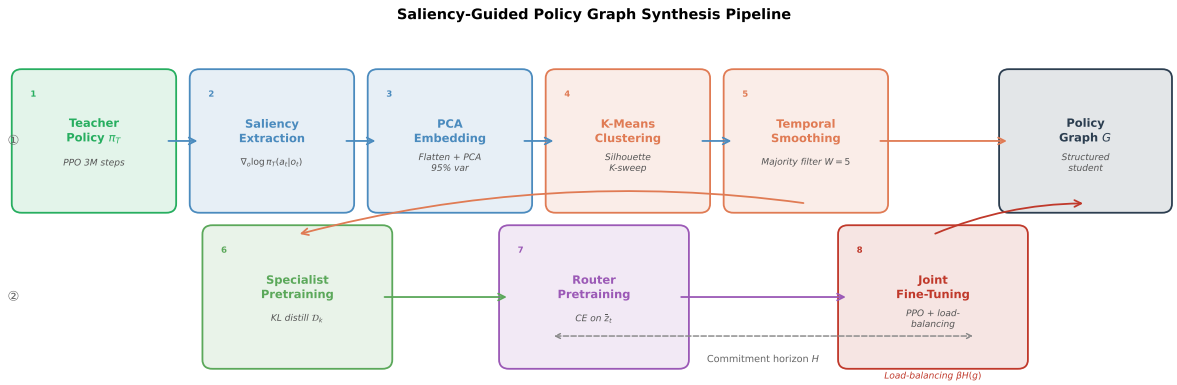


Figure 5.8: **Teacher-guided policy-graph synthesis from saliency traces.** A frozen teacher policy generates trajectories together with action-conditioned saliency maps. The saliency maps are embedded, clustered, and temporally smoothed into candidate behavioural regimes. Each regime becomes a specialist unit; the smoothed regime labels supervise a router. The resulting graph is then fine-tuned under the same commitment-bounded execution semantics introduced earlier in this chapter.

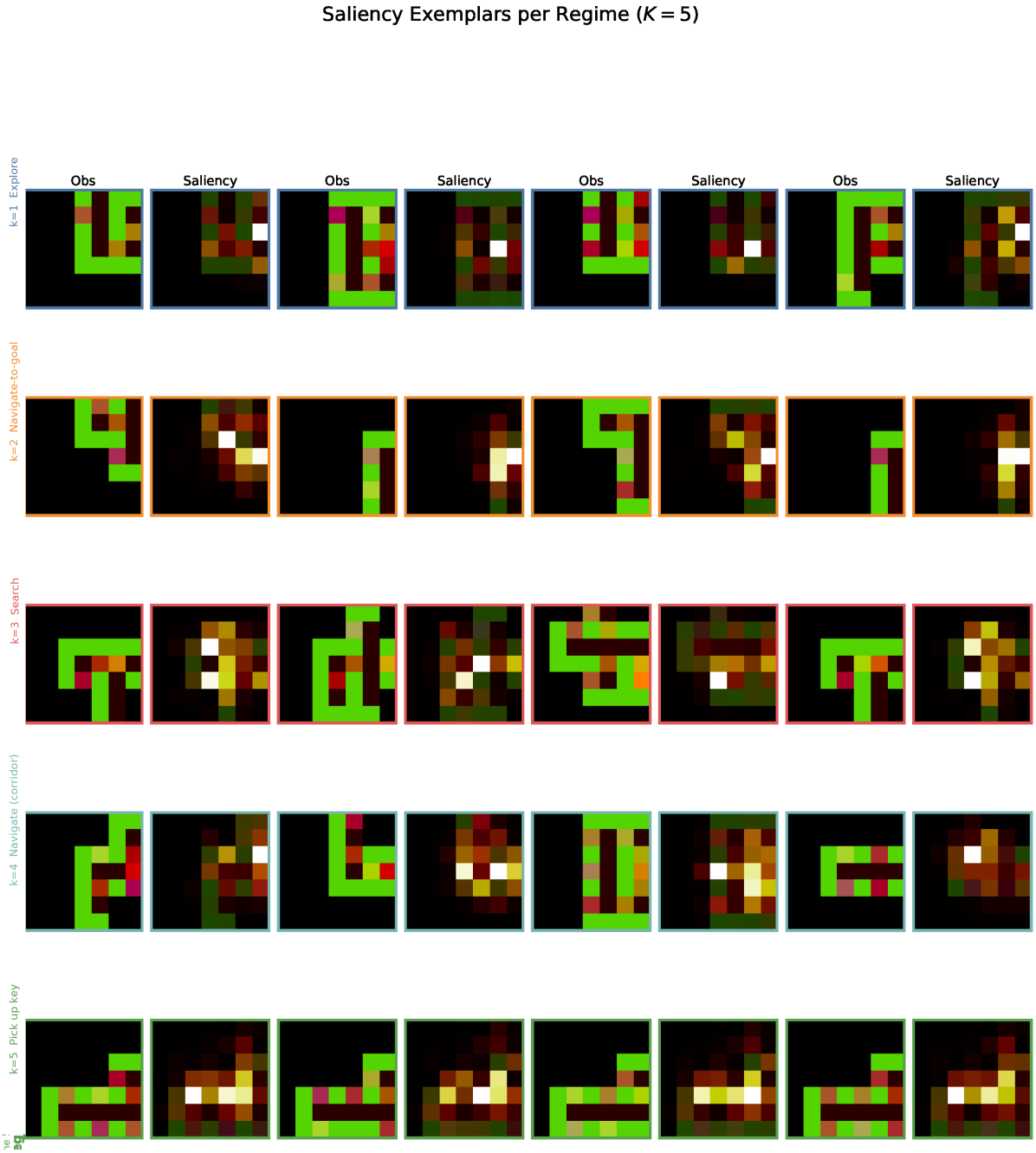


Figure 5.9: **Representative observations and action-conditioned saliency maps for the $K = 5$ KeyCorridor construction.** The discovered regimes correspond to recognisable phases such as room-entry exploration, search near the key, corridor navigation, key pick-up, and goal approach. The point is not that these labels are semantically perfect, but that the teacher repeatedly attends to different parts of the observation in different phases of behaviour.

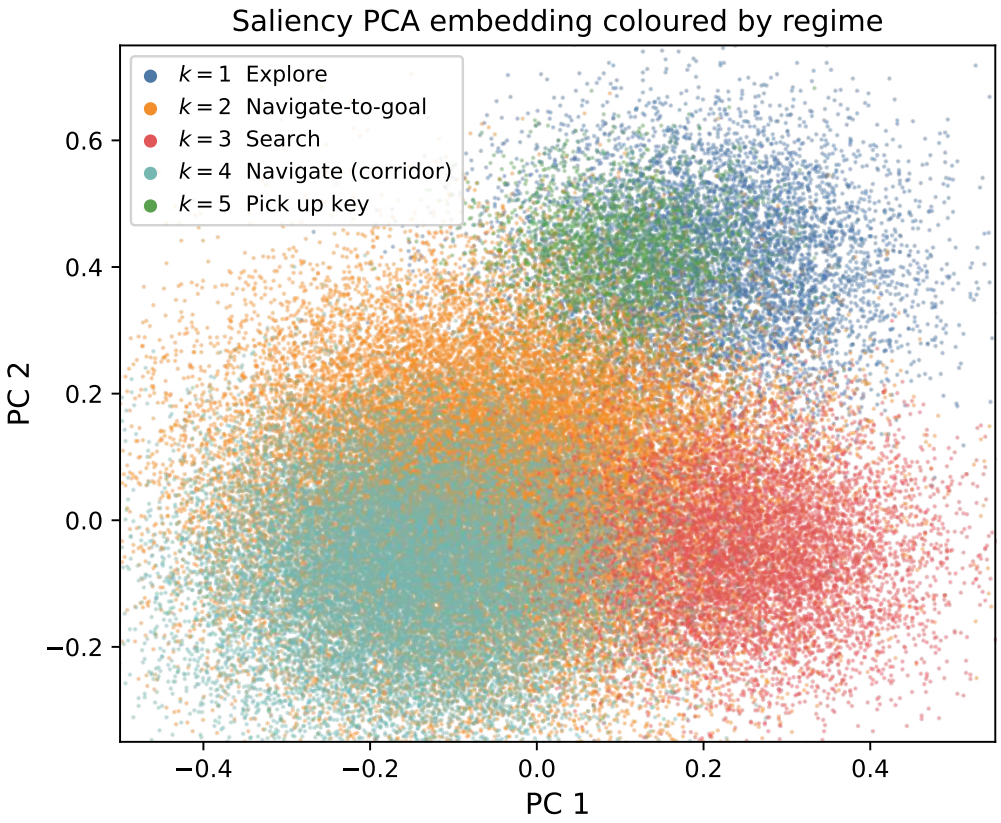


Figure 5.10: **Low-dimensional view of the KeyCorridor saliency embedding.** The first two PCA components do not separate the regimes perfectly, but they do reveal partially distinct regions in embedding space. The silhouette score on the full PCA-reduced representation is 0.295 for the silhouette-selected $K = 5$ construction.

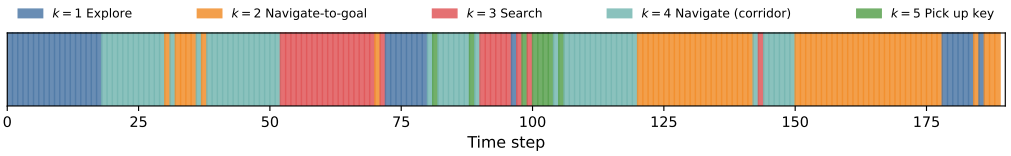


Figure 5.11: **Temporal regime trace for a representative KeyCorridor teacher episode.** The discovered labels persist over multi-step segments and align with recognisable phases of the task. Temporal smoothing suppresses brief assignment flicker near regime boundaries without removing the broader switching structure.

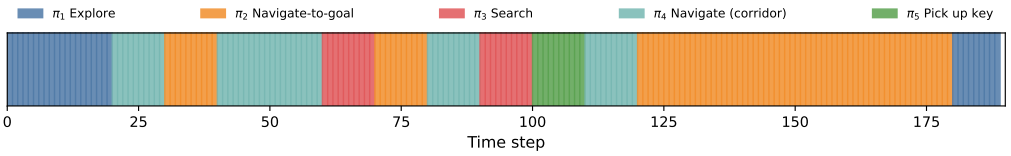


Figure 5.12: **Routing trace for the $K = 5$ saliency graph on a representative KeyCorridor episode.** Different specialists dominate distinct phases of behaviour, and the commitment horizon produces longer contiguous activations than the raw teacher regime labels. This is the key interpretability gain of the construction route: the student no longer behaves as a single opaque policy, but as a sequence of explicit unit activations.

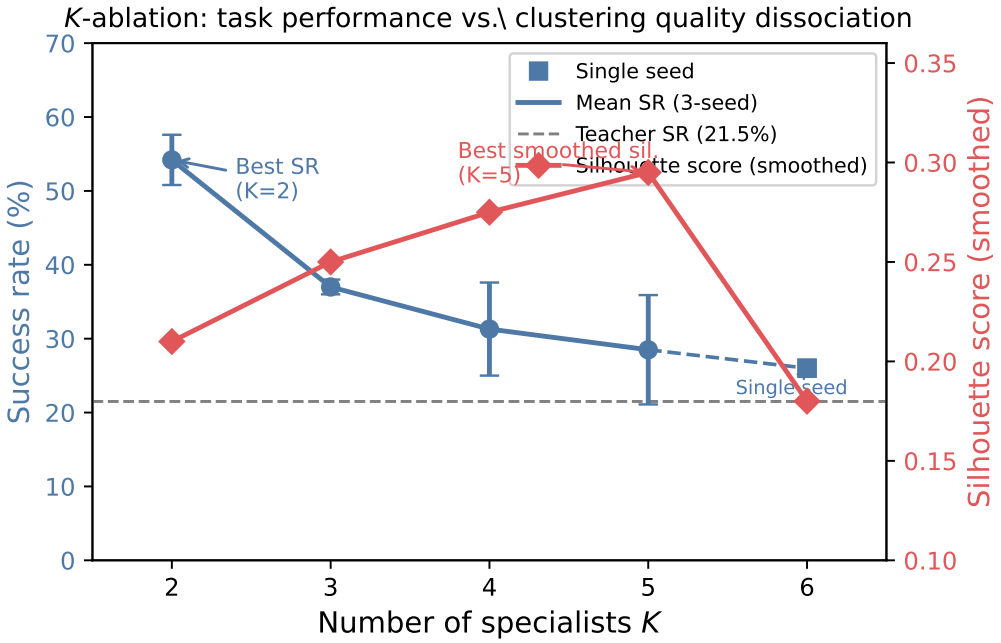


Figure 5.13: **K-ablation on KeyCorridorS3R3.** Downstream control performance peaks at $K = 2$, whereas the silhouette criterion peaks at $K = 5$. The dissociation is important: the silhouette-selected construction is the correct no-label result for this section, whilst the lower-K settings should be read as sensitivity analysis rather than as a replacement for the unsupervised route itself.

Chapter 6

Generalisations

Abstract

Real-world deployment demands that learned policies generalise beyond their training distribution—a requirement difficult to validate when benchmarks comprise dozens of manually designed tasks rather than diverse families of environments. This chapter introduces EnvCraft, a validation-first system that generates thousands of validated Gymnasium environments from natural-language concepts, enabling larger-scale generalisation studies than are usually practical with hand-built benchmarks. A multi-stage pipeline combines large language model code generation with automated testing and agent-based validation, producing environments that share a fixed observation-action interface whilst varying in dynamics, reward structures, and win conditions. Privileged agents with source-code access screen for unsuitable difficulty extremes and generate demonstration trajectories that bootstrap vision-based learning. Cross-validation experiments centred on procedurally generated Tetris variants provide within-family evidence that broader training distributions can improve performance on held-out tasks. This infrastructure addresses the scarcity of validated benchmark diversity identified in earlier chapters and provides a basis for future evaluation of whether modular systems genuinely generalise or merely overfit to narrow task distributions.

6.1 Introduction

Real-world deployment demands generalisation. Chapter 2 traced the division of labour from pin factories to flight computers, establishing that specialisation enables productivity gains only when workers—or policy units—transfer skills across contexts. Chapter 3

examined how the A320's flight computers, the French power grid's hierarchical control, and the Kangduo surgical robot's dual-console handover achieve reliability through architectural patterns: specialisation, redundancy, constrained transitions. Chapter 4 identified deployment challenges in sepsis treatment and telesurgery, revealing that learned policies must generalise beyond their training distribution whilst maintaining interpretability and bounded execution. The modular systems developed in Chapter 5—policy graphs with hard routing, commitment bounds, and distributed execution—inherit these principles. However, evaluating whether such systems genuinely generalise or merely overfit to narrow task distributions requires benchmark diversity at scales beyond existing suites.

Traditional RL benchmarks comprise dozens of manually designed tasks. The Arcade Learning Environment [127] standardised evaluation across Atari games; DeepMind Control Suite [128] provided continuous control tasks; Procgen [129] introduced procedurally generated levels. These contributions enabled algorithmic progress, yet benchmark scarcity creates a fundamental tension: as agents approach human-level performance on fixed suites, distinguishing genuine competence from task-specific memorisation becomes difficult. Procgen demonstrated that agents trained on limited level seeds catastrophically overfit when evaluated on held-out levels. NetHack [130] and Crafter [131] push complexity further, yet represent singular rule systems rather than diverse families of mechanics.

Prior approaches to environment diversity operate at different granularities. Procedural content generation varies layouts and textures within fixed rules. Automatic environment design methods such as POET [132] and PAIRED [133] co-evolve tasks and agents but rarely certify pixel-learnability. Game description languages like VGDL [134] enable compact specification but require bespoke tooling. What remains scarce is *validated diversity at the level of rules*—new dynamics, reward structures, and win conditions.

EnvCraft addresses this gap through a validation-first pipeline. Ideas become design briefs via gpt-oss-20b[1]; briefs become code via gpt-oss-120b[2]; code is tested and repaired; agent-based checks screen for degenerate cases. A privileged agent with full access to environment internals removes unsuitably difficult environments and generates demonstration data for bootstrapping vision-based policies. The final corpus comprises environments that are syntactically correct, API-compliant, and screened for obvious degeneracies—pixel-based learnability is not systematically verified.

Every EnvCraft environment exposes a fixed specification enabling cross-game training:

- **Observation**: 84×84×3 RGB array (uint8)
- **Action**: `MultiDiscrete([5,2,2])`—five movement options plus two binary but-

[1] `https://openai.com/index/introducing-gpt-oss/`
[2] `https://openai.com/index/introducing-gpt-oss/`

tons

- **Episode**: Maximum 1,000 steps
- **API**: Gymnasium-compliant with deterministic seeding

Representative examples of generated environments are shown in Figure 6.1.

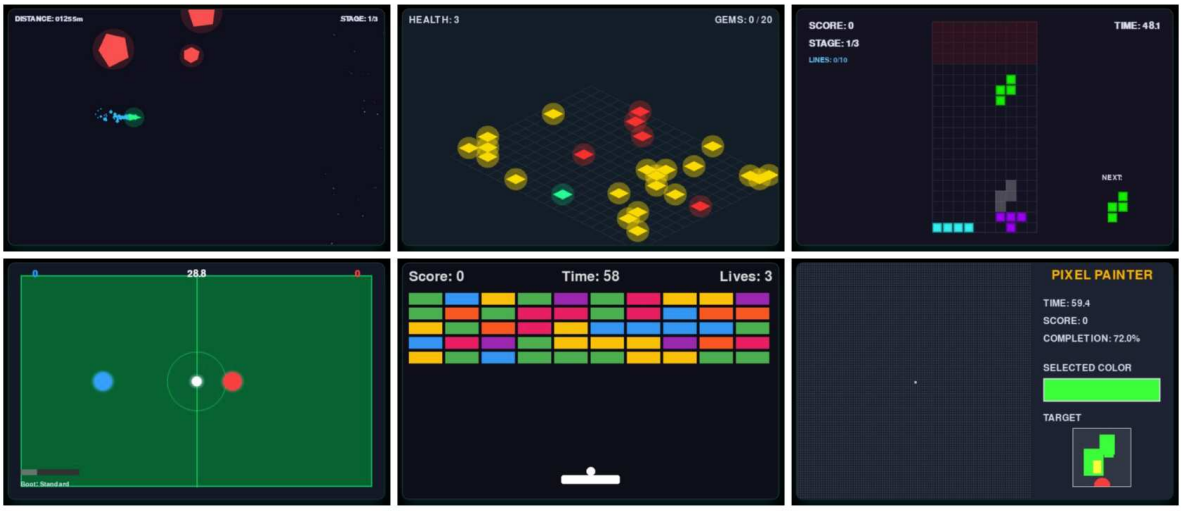


Figure 6.1: Example EnvCraft environments. Six representative rendered observations from distinct generated games, illustrating diversity in mechanics and visual style whilst sharing the fixed 84×84×3 RGB observation and `MultiDiscrete([5,2,2])` action interface.

The pipeline also generates privileged demonstration trajectories used to bootstrap vision-based learning; Section 6.4 describes this process.

This chapter makes three primary contributions:

1. A multi-stage validation pipeline producing 9,694 validated environments from 20,000 initial concepts (48.5% yield), incorporating privileged agent screening and privileged-rollout replay seeding for vision agent pretraining.
2. A privileged agent methodology that uses language model access to source code for difficulty screening and demonstration trajectory generation, bootstrapping vision-based learning via replay seeding and pretraining.
3. **Larger-scale within-family generalisation evaluation**: Using 1,000 procedurally generated Tetris environments with 10-fold cross-validation, this chapter demonstrates that broader training distributions produce significant positive transfer to held-out tasks: 68.7% of 1,000 environments show gains (7.4% mean improvement on split 0 as a representative example), with a monotonic relationship between training diversity and generalisation performance under this protocol.

6.2 Related Work

6.2.1 Benchmarks and Procedural Content Generation

The Arcade Learning Environment [127] established standardised evaluation across 57 Atari 2600 games, though subsequent work revealed issues with deterministic dynamics [135]. Mnih et al. [13] demonstrated that deep Q-networks could achieve human-level performance on many Atari games, whilst Rainbow [136] pushed performance further by combining multiple algorithmic improvements. DeepMind Control Suite [128] provided continuous control tasks with interpretable rewards. Whilst these suites enabled algorithmic progress, they prioritise consistency and reproducibility over rule-level diversity.

Recent benchmarks address overfitting through procedural content generation. Procgen [129] generates unlimited level variants within 16 fixed game types using deterministic seeds, demonstrating that agents trained on limited seeds catastrophically overfit when evaluated on held-out seeds. MiniGrid [119] provides gridworld navigation tasks with procedurally generated mazes and layouts, enabling studies of sample efficiency and partial observability. NetHack [130] wraps the classic roguelike game as a Gymnasium environment, offering extraordinary complexity through procedural dungeon generation, though the symbolic state representation and ASCII rendering differ substantially from pixel-based benchmarks. MiniHack [137] extracts NetHack mechanics into smaller, controllable tasks with faster episode turnover. Crafter [131] provides a single open-world survival game with procedurally generated terrain, evaluating agents through achievement-based metrics across diverse skills.

These approaches vary content—level layouts, terrain, entity placements—within fixed rule systems. Our work operates at a different granularity: we generate the rules themselves, producing entirely distinct game mechanics, reward structures, and win conditions whilst maintaining a common observation and action interface.

6.2.2 Automatic Environment Design

A growing body of work explores automatic generation of training environments to improve generalisation and robustness. POET [132] introduced open-ended co-evolution of environments and agents, progressively increasing difficulty through evolutionary selection whilst maintaining a diverse population of agent-environment pairs. PAIRED [133] frames environment design as an adversarial game in which an antagonist generates challenging environments whilst a protagonist learns to solve them, leading to robust zero-shot transfer. PLR [138] maintains a distribution over procedurally generated levels, prioritising replay of environments with high temporal-difference error to focus learning on the curriculum frontier. Quality-diversity methods such as MAP-Elites [139] search for diverse, high-performing solutions across behavioural dimensions, producing archives of

environments that cover different challenge characteristics.

These approaches verify learnability through agent training: environments that prove unlearnable within the training budget are discarded or down-weighted. However, this verification occurs *during* the training loop, requiring agents to attempt learning on potentially futile tasks. Our approach separates validation from training: privileged agents with state access provide rapid learnability probes before vision-based training begins, and the fixed interface enables independent validation once rather than per-training-run verification.

6.2.3 Language Models for Code Generation

Large language models now enable direct code synthesis from natural language [140]. Game description languages such as VGDL [134] provide declarative alternatives, but require bespoke interpreters; EnvCraft instead generates executable Gymnasium code directly, avoiding bespoke tooling whilst addressing correctness and learnability through automated testing and agent-based validation. Once environments are validated, DQfD-inspired replay seeding [141] provides an efficient path to bootstrapping vision-based policies from privileged-agent trajectories using standard temporal-difference objectives only (no supervised margin loss).

6.3 Code Generation Pipeline

The EnvCraft system decomposes environment creation into code generation and agent-based validation phases, implementing progressive refinement with empirical gates between stages. Figure 6.2 presents the complete pipeline architecture, which transforms natural-language game concepts into validated Gymnasium environments.

The first half of our system transforms natural-language concepts into executable Gymnasium environments through progressive refinement.

6.3.1 Concept Generation and Code Synthesis

The pipeline generates 20,000 diverse game concepts by sampling from curated pools comprising 42 genres, 56 mechanics, 51 themes, 36 twists, 19 mashups, and 30 experimental concepts. Four complementary strategies ensure broad coverage: genre-blend (combining three distinct genres), mechanical (assembling four individual mechanics), thematic (pairing visual theme with core mechanic and twist), and experimental (unusual single-concept games). Each idea specifies concrete mechanics, objects, win/loss conditions, and numerical parameters to ensure implementability. Deduplication via normalised text hashing removes exact duplicates whilst preserving meaningful variations.

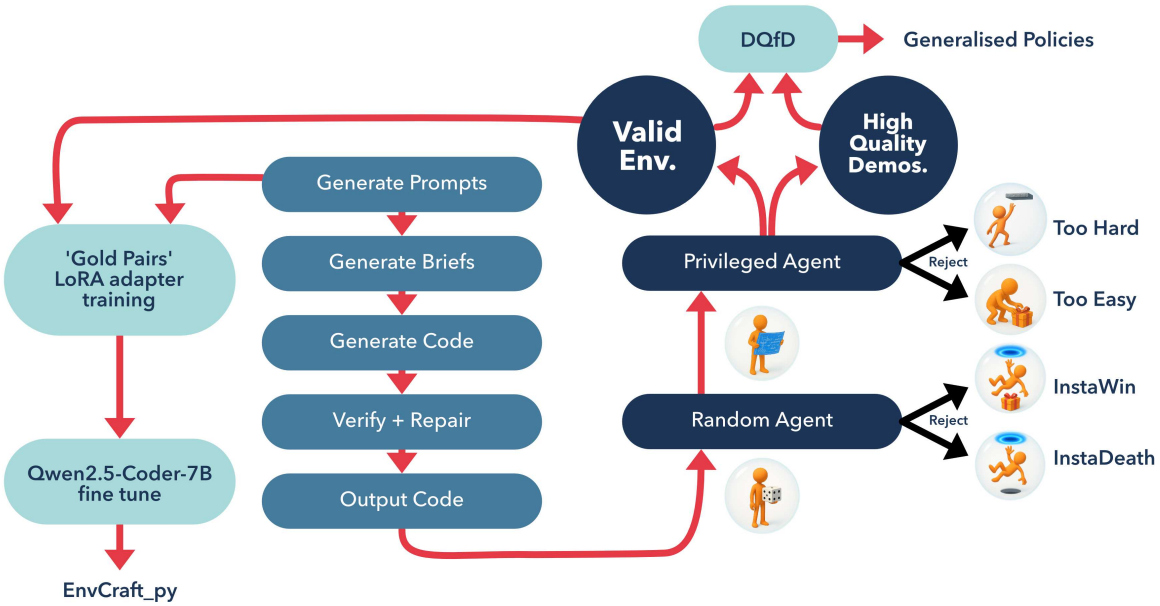


Figure 6.2: Complete EnvCraft pipeline. Game concepts progress through code generation (idea → brief → implementation → testing), random agent filtering (removing InstaWin/InstaDeath cases), and privileged rollout generation (difficulty assessment removes too-hard/too-easy cases; rollouts seed replay buffer for pretraining). Successfully validated environments are paired with their prompts as training data for fine-tuning code generation models.

These concepts are expanded into detailed 1,500–3,000 word design specifications using gpt-oss-20b, covering core mechanics, visual design, action space mapping to `MultiDiscrete([5,2,2`, reward structure, and termination conditions. An early viability check critiques each brief for internal consistency and implementability, filtering out specifications with impossible physics, contradictory win conditions, or action space mismatches. Of 20,000 initial ideas, 18,878 briefs (94.4%) pass this filter.

Validated briefs are transformed into executable Python code using gpt-oss-120b, generating complete 500–1,000 line Gymnasium [142] environments that implement the full API (`reset()`, `step()`, `render()`), produce 84×84×3 RGB observations, handle edge cases gracefully, and maintain deterministic behaviour under fixed seeding. This achieves a 94.9% success rate: 17,915 of 18,878 briefs produce syntactically valid, importable Python code. The 963 failures arise from malformed syntax, circular imports, or non-existent library references.

6.3.2 Testing and Repair

Generated code undergoes a comprehensive test suite covering syntactic correctness, API conformance, reinforcement learning invariants (bounded rewards, eventual termination, informative observations), and deterministic behaviour under fixed seeding. Of the 17,915 environments that pass code generation, 8,503 initially fail one or more tests. Rather than discarding these environments immediately, we implement an automated repair loop in which error messages, stack traces, and failing test descriptions are provided to a language model tasked with fixing the code whilst preserving the original design intent.

The repair process proceeds iteratively, with up to three attempts permitted per environment. The first repair pass successfully fixes 1,520 environments, representing 17.9% of the initial failures. Many of these are straightforward errors: incorrect variable references, off-by-one indexing mistakes, or missing imports. The second pass recovers an additional 701 environments (8.2% of failures), typically addressing more subtle issues such as edge cases in collision detection or state update ordering. The third and final pass fixes 192 environments (2.3% of failures), capturing a small number of complex multi-step repairs. In total, the iterative repair process recovers 2,413 environments that would otherwise have been lost.

Despite these efforts, 6,090 environments remain unfixable after three attempts and are discarded. Ultimately, 11,825 environments pass all code-level tests and proceed to agent-based validation. Figure 6.3 and Table 6.1 show the complete filtering cascade and exact counts at each stage.

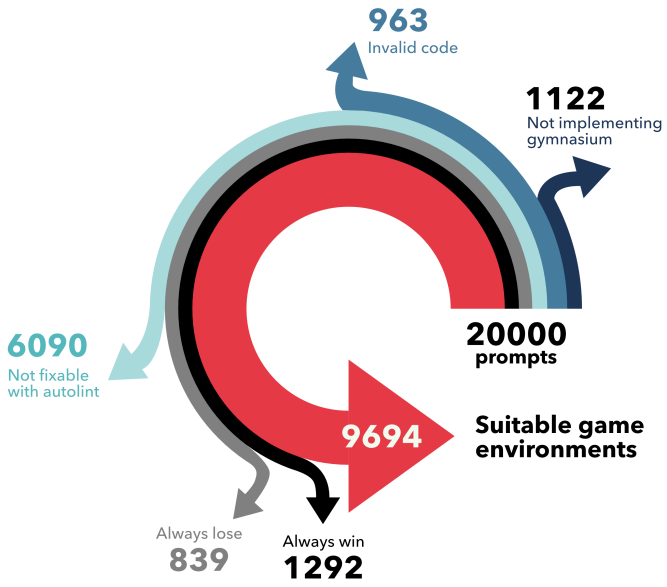


Figure 6.3: Environment filtering cascade. Starting from 20,000 generated game concepts, progressive filtering through design brief validation (18,878 pass), code generation (17,915 valid), automated testing with repair (11,825 pass after up to three repair iterations), and agent-based checks (9,694 final). Major losses occur during testing/repair (6,090 irreparable), random agent filtering (935 InstaWin + 606 InstaDeath), and privileged agent assessment (590 unsuitable difficulty). The 48.5% overall yield represents environments that are syntactically correct, API-compliant, and free of degenerate reward structures.

Table 6.1: Pipeline statistics showing input/output counts and pass rates at each stage.

Stage	Input	Output	Pass Rate
S1: Ideas	—	20,000	—
S2: Brief generation	20,000	18,878	94.4%
S3: Code generation	18,878	17,915	94.9%
S4: Test + repair	17,915	11,825	66.0%
S5: Random agent	11,825	10,284	87.0%
S6: Privileged agent	10,284	9,694	94.3%
Overall	20,000	9,694	48.5%

6.3.3 Random Agent Filtering

Environments that pass code-level testing may nonetheless exhibit degenerate behaviours that render them unsuitable for reinforcement learning research. Two baseline agent checks are applied to identify and eliminate such edge cases.

The first check, InstaWin detection, executes a random policy for multiple episodes and monitors the reward distribution. Environments in which random actions consistently achieve high returns—indicating that success requires no learning whatsoever—are flagged as degenerate. Such environments typically arise from overly generous reward shaping, trivial win conditions, or bugs that inadvertently reward all actions equally. This check removes 935 environments from the corpus.

The second check, InstaDeath detection, verifies that a no-op policy (an agent that takes no actions) does not immediately fail. Whilst it is acceptable for a no-op agent to eventually lose by timeout, instant death without any agency indicates unavoidable failure states that make learning impossible. These failures often stem from spawn-point collisions, initial conditions that violate game constraints, or aggressive enemies that attack before the agent can react. This check removes an additional 606 environments.

Following these random agent filters, 10,284 environments remain and proceed to privileged agent evaluation. These environments are syntactically correct, API-compliant, and free of the most obvious degeneracies, though they have not yet been verified for learnability from pixel observations.

6.4 Privileged Rollout Generation

"Demonstrations" here means trajectories collected from a non-learning policy and used to seed or supervise a vision-based learner. In EnvCraft, demonstrations are generated automatically by the privileged code-access policy rather than collected from humans or external datasets. All vision-based training uses privileged-rollout replay seeding and a pretraining phase: the replay buffer is initialised with privileged rollouts, the vision agent is pretrained by sampling exclusively from this seeded replay, then continues with standard online training using temporal-difference objectives only (no supervised margin loss as in canonical DQfD [141]); code access is never available to the learner.

Passing code-level tests and random agent checks does not guarantee an environment is suitable for reinforcement learning. A game might execute correctly yet be unlearnable due to hidden state dependencies, adversarial dynamics, or reward structures that require capabilities beyond current algorithms. We address this through a privileged agent that has access to information unavailable to a standard vision-based learner.

6.4.1 Privileged Policy Synthesis

For each environment that passes random agent filtering, we synthesise a privileged policy using gpt-oss-120b with read-only access to the environment's source code. The model analyses the game logic, state representation, reward function, and termination conditions, then generates a Python policy class with an `act(state)` method that maps internal game state to actions.

This privileged agent operates on the complete internal state—player positions, enemy locations, item inventories, timers, and any other variables defined in the code—rather than the 84×84×3 rendered observation. The agent "plays" the environment in a read-only capacity: it observes the full state at each timestep and selects actions, but cannot modify any game variables directly. This asymmetry is intentional: the privileged agent provides a high-performing baseline representing what could be achieved with complete state information.

6.4.2 Difficulty Assessment

The privileged agent serves as a pragmatic screening heuristic for environments at difficulty extremes. The synthesised privileged policy is executed for multiple episodes and the outcome distribution is analysed under the rollout budget. If the privileged agent—with full state access—cannot consistently achieve positive outcomes, the environment may have design issues that make it unsuitable for our benchmark: potentially impossible win conditions, adversarial dynamics, or reward structures with no readily discoverable optima. Whilst the privileged agent is not guaranteed optimal and may fail for reasons unrelated to intrinsic unsolvability, environments it cannot solve are unlikely to provide useful learning signal for vision-based policies and are removed from the corpus. Conversely, if the privileged agent achieves maximum possible performance with near-certainty, the environment may lack meaningful challenge or have degenerate solutions accessible even to simple heuristics. Whilst not as problematic as potentially unsolvable games, trivially easy environments (as assessed by this heuristic) provide limited value for evaluating agent capabilities and are likewise removed.

This pragmatic filtering removes 590 environments from the 10,284 candidates, yielding a final corpus of 9,694 environments. This heuristic may exclude some learnable environments (where the privileged agent fails but vision agents might succeed) and retain some poorly-designed ones (where the privileged agent succeeds by exploiting structure unavailable to vision agents); it biases the corpus towards environments whose challenge is visible to a code-access policy, and may therefore under-represent tasks where perceptual difficulty is the dominant obstacle.

6.4.3 Privileged Rollout Generation and Replay-Seeded Pretraining

For all 9,694 environments, the privileged agent executes extended rollouts and complete trajectories are recorded as tuples:

$$\mathcal{D} = \{(o_t, a_t, r_t, o_{t+1}, d_t)\}_{t=1}^{T} \tag{6.1}$$

where o_t is the *rendered* 84×84×3 observation (not the internal state), a_t is the action selected by the privileged policy (based on internal state), r_t is the reward, o_{t+1} is the next observation, and d_t is the termination flag.

These demonstrations have a distinctive property: the actions are informed by information not present in the observations. A vision-only agent must infer, from pixel patterns alone, the action choices that the privileged agent made using complete state knowledge — the demonstrations thus encode implicit information about what visual features correlate with high-performing behaviour. For the generalisation experiments, 1,000 transitions per environment seed the replay buffer; complete pretraining and online training details are provided in Section 6.5.

6.5 Generalisation Experiments

The scale of the corpus—9,694 validated environments—enables experimental designs that are rarely practical with hand-built benchmarks. Tetris is chosen as the primary evaluation domain because its objective is unambiguous: longer episodes are always better, irrespective of the underlying reward scale. The 1,000 generated Tetris environments differ markedly in board geometry, block distributions, gravity schedules, termination rules, and reward shaping, making raw episode lengths incomparable across environments; episode length relative to a random baseline provides a clean, monotonic, per-environment metric.

6.5.1 Experimental Protocol

One thousand distinct Tetris environments were procedurally generated within the EnvCraft framework and randomly partitioned into ten folds of 100 environments each. For each cross-validation split, one fold serves as the test set whilst the remaining 900 environments form the training set, yielding ten disjoint train–test partitions. This 900/100 split achieves two aims: (i) it provides sufficient diversity during training to support non-trivial generalisation, and (ii) it furnishes a held-out panel of 100 genuinely unseen environments on which to assess out-of-distribution performance. Each environment appears in the test set exactly once across the ten folds, providing 1,000 environment-level

generalisation measurements.

The agent uses a Duelling Deep Q-Network architecture [143] with approximately 12 million parameters, processing 84×84×3 RGB observations through a five-layer convolutional backbone before splitting into separate value and advantage streams. Double DQN [144] with prioritised experience replay [145], ϵ-greedy exploration (annealed from 1.0 to 0.1), Adam optimisation [8] (learning rate 3×10^{-4}), and n-step returns ($n = 3$, $\gamma = 0.99$).

The training protocol (held constant across all diversity conditions) proceeds as follows:

- **Replay seeding**: 1,000 transitions per training environment seed the replay buffer before any learning begins (10,000 collected per environment; the remainder are available for extended runs).
- **Pretraining**: The vision agent is pretrained for 250,000 gradient updates, sampling minibatches exclusively from the seeded replay buffer, before any online interaction.
- **Online training**: Standard Double/Duelling DQN with prioritised experience replay, sampling from the replay buffer containing both privileged-generated and agent-generated transitions.

The training curriculum cycles through the 900 training environments in shuffled order, with the agent experiencing 10,000 steps per environment before rotating to the next, yielding approximately 9 million total environment interactions per cross-validation split. Gradient updates occur every four environment steps.

For each held-out environment, mean episode lengths under the trained and random policies are estimated from 1,000 episodes each (reducing Monte Carlo noise), with standard errors based on empirical episode-length variance. The 10-fold design ensures no individual environment dominates the evaluation and that results are not artefacts of a particular train–test partition.

6.5.2 Results and Analysis

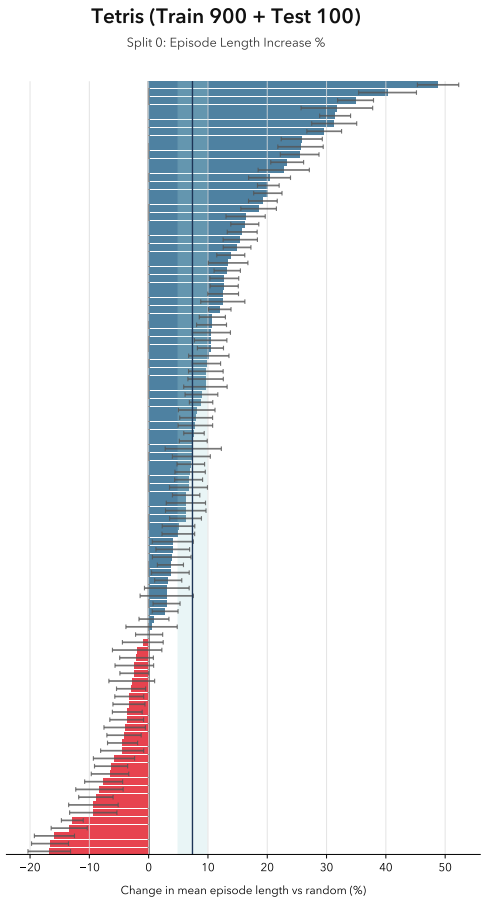
Performance is measured as the percentage change in mean episode length of the trained policy relative to a random policy, normalised per environment so that each environment contributes equally to the aggregate statistics regardless of absolute episode scale.

Across all ten folds, 687 of 1,000 environments (68.7%) show positive transfer from the trained policy over the random baseline. The overall mean improvement is approximately 1.96 steps (standard deviation 3.92 steps), indicating that training on 900 heterogeneous Tetris variants induces a systematic generalisation benefit despite considerable variability across individual environments. Figure 6.4a illustrates split 0 as a representative example:

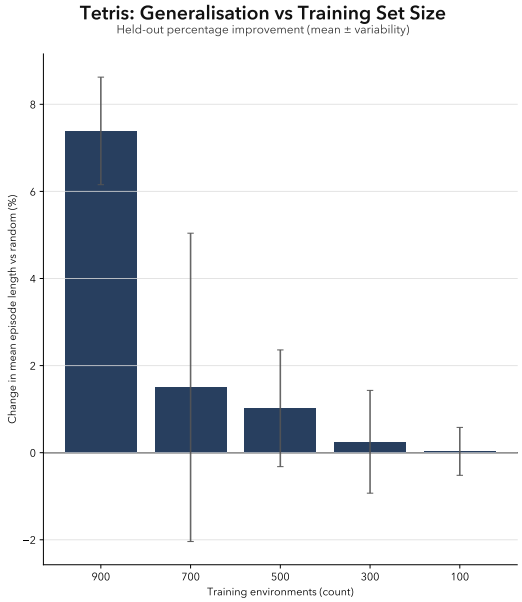
the mean improvement is 7.4% (95% CI [4.97%, 9.81%]), with the confidence band lying wholly to the right of zero confirming statistically significant positive transfer. Figure 6.4b shows the scaling behaviour across diversity conditions.

6.5.3 Scaling with Training Diversity

To assess scaling, policies were trained on subsets of 700, 500, 300, and 100 environments using the same evaluation framework. Reduced-diversity conditions use fewer cross-validation splits (five for the 700- and 500-environment conditions; one each for 300 and 100), so these results are indicative rather than a precise estimate of a change-point. Figure 6.4b shows a monotonic trend: as the number of training environments decreases, the mean generalisation effect degrades markedly, reaching near-zero in the 100–300 settings. This confirms that the generalisation benefit reflects broad environmental coverage during training rather than a few privileged environments.



(a) Generalisation to held-out environments. Each horizontal bar represents percentage improvement in mean episode length for one of 100 test environments (split 0), sorted by performance. The solid vertical line (7.39%) marks mean improvement; shaded band shows 95% CI [4.97%, 9.81%]. Error bars indicate per-environment 95% CIs. The majority of environments show positive transfer.



(b) Generalisation decreases with reduced training-set diversity. Mean percentage improvement in episode length as training set size varies (100, 300, 500, 700, 900 environments). Error bars show standard deviations across test environments and cross-validation splits (note: low-diversity conditions use fewer splits).

Figure 6.4: **Tetris generalisation experiments.** Training on 900 diverse Tetris variants produces statistically significant positive transfer to held-out environments (panel a). Generalisation decreases with reduced training diversity, with substantially lower transfer in low-diversity settings (panel b).

6.6 Discussion

The pipeline has inherent constraints: the fixed action space `MultiDiscrete([5,2,2])` excludes continuous-control settings; visual style is biased towards 2D arcade games; the three-attempt repair cap leaves complex multi-object interactions as the primary remaining failure mode. The privileged agent provides a useful heuristic but may not find optimal strategies for all games, biasing the corpus towards environments whose challenge is visible to a code-access policy. The Tetris generalisation results are honest about their scope: statistically significant transfer (68.7% of 1,000 environments, 1.96-step mean improvement) with modest effect sizes and high variability, all within a single game family. The core empirical claim is a within-family result; cross-domain generalisation remains an open question, and the benchmark is primarily a tool for making that question tractable at larger scale than existing hand-built suites allow.

6.7 Conclusion

This chapter presents EnvCraft, a validation-first system producing 9,694 diverse, validated Gymnasium environments from 20,000 initial concepts. The pipeline combines code generation with multi-stage agent-based filtering: random agents eliminate degenerate cases, whilst privileged agents with source code access screen for difficulty extremes and generate demonstration trajectories for replay seeding and pretraining of vision-based policies.

Large-sample within-family generalisation experiments on 1,000 procedurally generated Tetris environments show statistically significant positive transfer across all ten cross-validation folds: 68.7% of environments show gains, with a mean improvement of approximately 1.96 steps overall. Scaling experiments confirm a monotonic relationship between training diversity and generalisation performance, with near-zero transfer in the lowest-diversity settings. These results demonstrate that the benchmark supports generalisation studies at a scale still uncommon in reinforcement learning, whilst leaving cross-domain generalisation as an open question.

The complete corpus, interactive exploration tool, and open-source library are available at `https://experiments.standardrl.com/envcraft`.

Chapter 7

Models

Abstract

Distributed policy graphs—grounded in the division-of-labour principles discussed in Chapter 2 and formalised in Chapter 5—require efficient edge models to make real-world deployment practical. When policy units are distributed across heterogeneous hardware, the computational cost and communication overhead of transmitting high-dimensional visual observations can dominate decision latency, particularly on resource-constrained edge devices.

This chapter introduces MiniConv, a library of small convolutional encoders designed to compile cleanly to OpenGL fragment shaders for broad embedded GPU support. A split-policy architecture is realised in which a lightweight on-device encoder extracts compact visual features that are transmitted to a remote policy head, reducing decision latency in bandwidth-limited settings and lowering server-side compute per request. Across three visual control tasks trained with PPO, SAC, and DDPG, MiniConv encoders remain competitive with the chapter's Full-CNN baselines under pixel observations in the reported fixed-seed runs, whilst enabling practical deployment on devices ranging from the Raspberry Pi Zero 2 W to the NVIDIA Jetson Nano. The infrastructure developed here directly supports the distributed policy graph deployment discussed in Chapter 8.

7.1 Introduction

Policy graphs—formalised in Chapter 5—embody the division of labour identified in Chapter 2: specialist policy units coordinate through hard routing and commitment bounds, inheriting the architectural patterns that enable A320 flight computers and power

grid controllers to achieve reliability through modularity. Chapter 5 motivates a deployment picture in which rapid, reactive components execute on low-power edge devices near actuators, whilst more deliberate reasoning can run on remote GPU clusters. This distributed execution exploits heterogeneous hardware—edge processors handle time-critical perception, cloud servers handle optimisation—whilst commitment bounds limit handoff frequency to control communication overhead.

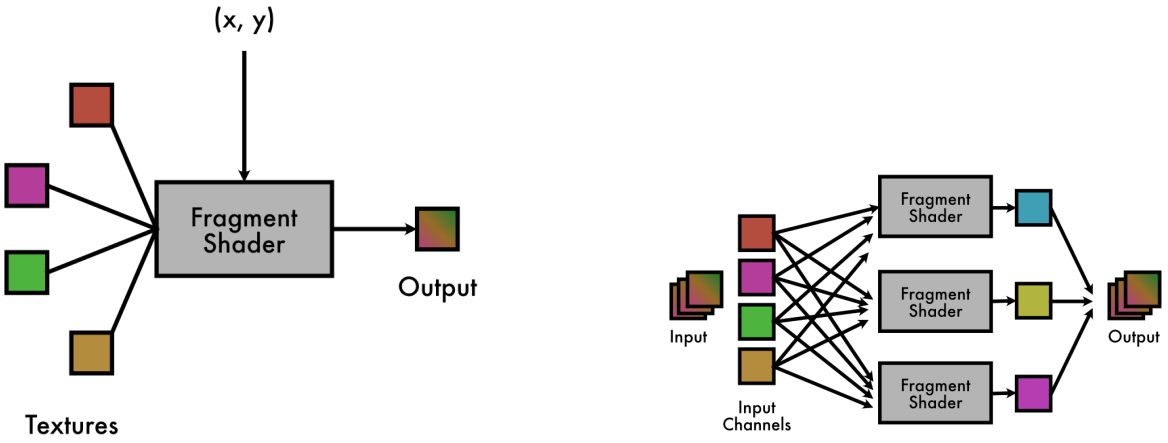
However, the practical viability of this architecture depends critically on edge efficiency. When a policy unit processes high-dimensional visual observations on resource-constrained hardware—a Raspberry Pi Zero 2 W with 512 MB RAM and an embedded Broadcom GPU, for instance—two bottlenecks emerge: the computational cost of feature extraction and the communication cost of transmitting observations. A conventional deployment transmitting full RGB frames to a remote server incurs substantial decision latency in bandwidth-limited settings and concentrates compute load centrally. Conversely, an encoder small enough to run efficiently on-device reduces both communication overhead and server-side load, enabling the edge-to-cloud division of labour that policy graphs require.

This chapter introduces MiniConv, a library of compact convolutional encoders designed for this deployment context, and evaluates the resulting split-policy architecture across learning performance, on-device execution, decision latency, and server scalability. The findings establish that lightweight visual encoders can serve as components of distributed policy graphs—supporting the edge-to-cloud architectures explored in Chapter 8.

7.2 Related Work

Approaches to on-device neural network inference range from specialist hardware accelerators [146] to architectures such as MobileNet [147] that achieve favourable accuracy–efficiency trade-offs through depthwise separable convolutions, and post-hoc compression methods (pruning, quantisation, and knowledge distillation), surveyed in [148].

More directly related to split-policy execution, several systems partition deep neural network inference between end devices and the edge or cloud to optimise latency and resource usage under bandwidth constraints. Neurosurgeon [47] selects partition points in DNNs to balance device computation against transmission cost, whilst Edge Intelligence [149] explores on-demand co-inference with device–edge synergy. Teerapittayanon *et al.* [150] consider distributed DNN execution across end devices, edge servers, and the cloud. MiniConv is complementary: it applies a similar division of labour to RL policies, emphasising wide hardware support through OpenGL shader execution and transmitting compact feature representations rather than raw observations. This work evaluates the resulting trade-offs in decision latency, scalability, and device resource pressure.



(a) Fragment shader input/output.

(b) Mapping CNN layers to shader passes.

Figure 7.1: OpenGL fragment shaders can implement convolution and pooling by sampling input textures and writing output textures.

7.3 Implementation

The *MiniConv library* provides small, composable encoder blocks designed to compile cleanly to OpenGL fragment shaders, respecting practical constraints such as texture binding and sampling limits. MiniConv encoders are instantiated here with K output channels (specifically $K = 4$ and $K = 16$) and trained end-to-end together with a downstream policy in PyTorch. At deployment, only the MiniConv encoder runs on-device (via OpenGL), producing a K-channel feature tensor per frame; only this tensor is transmitted to the server-side policy head. MiniConv is a *library* rather than a single fixed architecture: K and block compositions can be varied to meet device and bandwidth constraints.

The on-device encoder is deployed using OpenGL fragment shaders, which compute each output pixel as a function of one or more input textures and are widely supported across embedded GPUs. This execution model maps naturally to convolution and pooling: a shader samples a neighbourhood of an input texture and writes an output texture, as illustrated in Figure 7.1. MiniConv exploits this mapping whilst respecting the practical limits of low-cost devices. For example, on the Raspberry Pi Zero 2 W, fragment shaders can sample from a maximum of eight bound textures, and each shader is subject to a finite sampling budget (64 texture samples in our deployment). Since each shader pass outputs four channels (RGBA), encoders with larger K are implemented via multiple passes. These constraints inform the choice of kernel sizes, channel packing, and layer compositions used by MiniConv.

7.4 Evaluation

Deploying split-policy RL on edge devices requires that the on-device encoder preserves policy performance whilst respecting strict compute, memory, and power constraints. The evaluation is organised around eight practical questions:

Q1 Does a split-policy architecture match the learning performance of a conventional Full-CNN baseline under visual observations?

Q2 Does the compressed on-device representation retain sufficient task-relevant information to support high-return behaviour?

Q3 How do per-frame inference latency and variability change under sustained on-device execution?

Q4 What memory footprint does on-device inference impose, and how much RAM headroom remains for other tasks?

Q5 What is the effect of sustained inference on device thermal state and throttling behaviour?

Q6 At what link bandwidth does split inference reduce end-to-end decision latency relative to transmitting full observations?

Q7 On low-power devices, how does OpenGL shader execution compare to a CPU implementation in throughput and stability?

Q8 How do power limits and power consumption affect inference throughput and stability?

These questions are addressed through learning experiments on visual control tasks, on-device execution benchmarks, and end-to-end measurements of decision latency and server scalability under bandwidth constraints.

7.4.1 Learning

MiniConv encoders are evaluated on two MuJoCo locomotion tasks (*Walker2d*, *Hopper*) and the classic control *Pendulum* task under visual observations. Walker2d is trained with PPO [17], Hopper with SAC [18], and Pendulum with DDPG [73], selected based on preliminary stability under pixel observations and standard practice in Stable-Baselines3 for the respective tasks. Unless otherwise stated, *Walker2d* and *Hopper* are trained for 2,000 episodes and *Pendulum* for 1,000 episodes. Because algorithms differ across tasks, cross-task comparisons are not meaningful; the focus is on within-task comparisons between encoders. Results are reported for a single run per condition (fixed seed), and variance across seeds is not yet characterised.

Table 7.1: Algorithms used for each visual control task.

Task	Algorithm	Selection rationale
Walker2d-v4	PPO	On-policy baseline that trained without collapse under pixel observations in our experimental configuration.
Hopper-v4	SAC	Common off-policy baseline for continuous control that trained without collapse under pixel observations in our experimental configuration.
Pendulum-v1	DDPG	Lightweight deterministic baseline that trained without collapse for Pendulum under pixel observations in our experimental configuration.

Algorithms and baselines

Table 7.1 summarises the learning algorithm used for each task.

For each task, the Full-CNN baseline corresponds to the default convolutional feature extractor used by Stable-Baselines3 [151] for image observations (`CnnPolicy`). The MiniConv conditions replace only this observation encoder (with $K \in \{4, 16\}$ output channels); the downstream policy, value networks, and all other training settings are unchanged across encoder variants. The split-policy architecture does not assume a particular RL algorithm; results should be interpreted as within-task evidence that encoder partitioning can be compatible with learning under multiple common RL algorithms.

All experiments use 84×84 RGB pixel observations stacked over three frames, processed through SB3's default image normalisation. Environments use Gymnasium [152]: *Walker2d-v4* and *Hopper-v4* via MuJoCo [153], and *Pendulum-v1* (Classic Control).

These experiments test whether replacing the standard image encoder with MiniConv preserves the ability to learn high-return behaviour under pixel observations. Within each task, MiniConv remains competitive with the Full-CNN baseline, but summary statistics exhibit task- and representation-size-dependent trade-offs between final and mean return. Each condition reports Best (maximum episodic return observed), Mean (average episodic return over training), and Final (mean episodic return over the final 100 episodes). These findings address Q1–Q2. Given that each condition is evaluated in a single fixed-seed run, the reported differences should be interpreted as indicative rather than statistically characterised.

Walker2d (PPO)

MiniConv $K = 4$ achieves slightly higher final return than Full-CNN (3360 vs 3296), whilst Full-CNN attains higher mean return over training (2800 vs 2680). $K = 16$ reaches the highest single episode (3800) but exhibits lower sustained performance, suggesting less consistent behaviour under pixel observations (Table 7.2).

Table 7.2: Walker2d (PPO): episodic return statistics over 2,000 episodes (single fixed-seed run).

Architecture	Best	Final	Mean	Episodes
MiniConv encoder (K=4)	3640	3360	2680	2000
MiniConv encoder (K=16)	3800	3184	2320	2000
Full-CNN	3600	3296	2800	2000

Table 7.3: Hopper (SAC): episodic return statistics over 2,000 episodes (single fixed-seed run).

Architecture	Best	Final	Mean	Episodes
MiniConv encoder (K=4)	2680	2360	1680	2000
MiniConv encoder (K=16)	2640	2200	1600	2000
Full-CNN	2656	2240	1720	2000

Hopper (SAC)

MiniConv $K = 4$ yields the strongest final return on Hopper (2360 vs 2240 for Full-CNN), whilst Full-CNN attains higher mean return (1720 vs 1680). The gap between best and final return across all encoders indicates substantial variability in sustained performance under pixel observations in these single-seed runs (Table 7.3).

Pendulum (DDPG)

Both MiniConv encoders outperform Full-CNN on Pendulum final return ($K = 16$: -180 vs -248 for Full-CNN), consistent with this task's sensitivity to smooth, consistent control (Table 7.4). The improvement of $K = 16$ over $K = 4$ suggests that a richer transmitted representation benefits tasks where representation quality affects stability.

Taken together, these results suggest that MiniConv encoders can remain competitive with a conventional Full-CNN baseline under visual observations, but do not uniformly dominate across summary statistics. Encoder-4 achieves slightly higher final return on *Walker2d* and *Hopper*, whilst Full-CNN attains the higher mean return in both tasks; encoder-16 is less effective on the locomotion tasks but performs best on *Pendulum*. This pattern indicates that the appropriate representation size is task-dependent and should be selected alongside device compute and bandwidth constraints.

7.4.2 Execution Performance

Per-frame inference time is characterised as a function of input size and device class; drift under sustained load is evaluated; and CPU temperature, RAM utilisation, and power consumption are recorded. These experiments address Q3–Q5, Q7, and Q8. The computation–communication trade-off underpinning split inference is then analysed to address Q6.

Table 7.4: Pendulum (DDPG): episodic return statistics over 1,000 episodes (single fixed-seed run).

Architecture	**Best**	**Final**	**Mean**	**Episodes**
MiniConv encoder (K=4)	-140	-192	-244	1000
MiniConv encoder (K=16)	-136	-180	-232	1000
Full-CNN	-142	-248	-288	1000

In addition to task-scale inputs, a high-resolution stress test (up to 3000×3000) is included to expose throttling and power-limit behaviour under sustained load, particularly on the Jetson Nano.

Figure 7.2 summarises per-frame processing time across devices as the input size varies. As the input size increases, frame processing time increases on the Raspberry Pi platforms, whilst the Jetson Nano exhibits substantially lower times across the tested range. On the Pi Zero 2 W, maintaining a frame rate of five frames per second requires keeping the input size below roughly 500 pixels per side (that is, below 500×500).

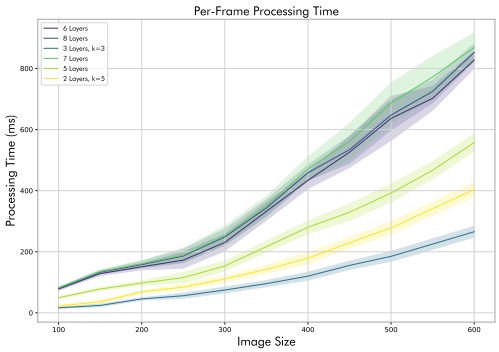
Sustained inference time is measured over extended runs (Figure 7.3). The Jetson Nano exhibits a marked increase in per-frame time after an initial period, and power limits alter this behaviour. For the Pi Zero 2 W, GPU (OpenGL) inference is substantially faster and more stable than CPU (PyTorch) inference over the same horizon.

To characterise the resource pressures associated with sustained inference, Figure 7.4 reports CPU temperature and RAM utilisation on the Pi Zero 2 W (CPU vs GPU execution), and power usage and memory pressure on the Jetson Nano (5W cap vs no limit). Across these experiments, RAM utilisation remains comparatively stable, whilst temperature and power reflect the expected constraints of sustained on-device execution.

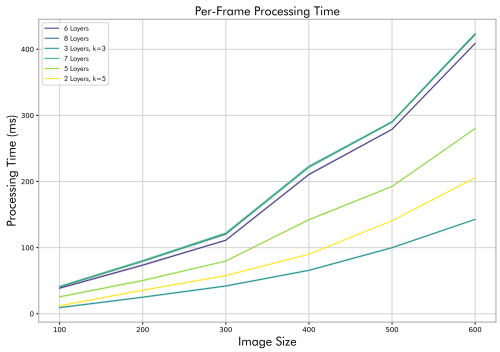
Ultimately, the utility of split-policy execution depends on the balance between computation and communication. Figure 7.5 illustrates the decision-latency components that vary between a server-only pipeline and the split-policy pipeline.

A simplified bandwidth model considers B as link bandwidth in bits per second, X the input width and height, n the number of stride-two layers in the on-device encoder (so the transmitted feature map has spatial size $(X/2^n) \times (X/2^n)$), and j the per-frame on-device processing time. Both raw observations and encoded features are transmitted as uncompressed uint8 buffers: a full RGBA frame requires $4X^2$ bytes, whilst a K-channel feature map requires $K(X/2^n)^2$ bytes ($K = 4$ for the latency experiments). Image compression would shift the break-even point and is left to future work. Server-side compute is excluded to isolate the communication break-even point; server-side compute reductions are evaluated separately in the scalability experiment. Under these assumptions, split-policy inference yields a lower decision latency than a server-only pipeline when:

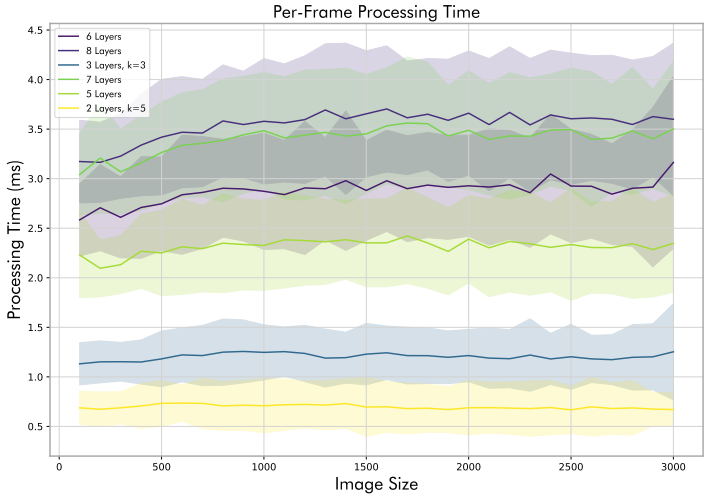
$$B < \frac{32X^2\left(1 - \frac{K}{4 \cdot 2^{2n}}\right)}{j}.$$



(a) Raspberry Pi Zero 2 W.



(b) Raspberry Pi 4B.



(c) NVIDIA Jetson Nano.

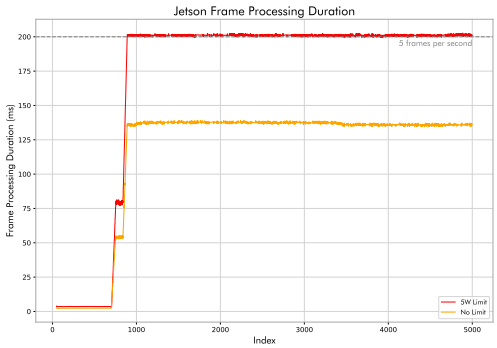
Figure 7.2: Per-frame processing time across devices as the input image size varies (mean of 100 consecutive inferences; shaded region shows standard deviation).

For the Pi Zero 2 W configuration in Figure 7.3b ($X = 400$, $n = 3$, $j \approx 0.1s$, $K = 4$), this yields a break-even bandwidth of approximately $50.4\,\mathrm{Mb\,s^{-1}}$.

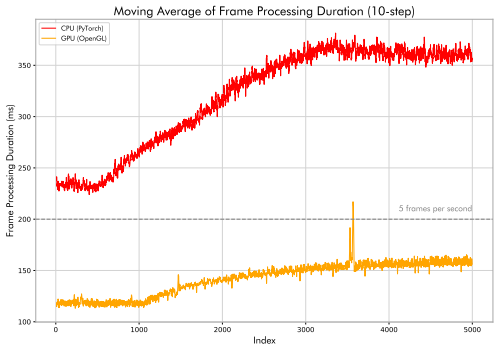
7.4.3 End-to-End Decision Latency

To address Q6 empirically, end-to-end *decision latency* is measured as the median wall-clock time (over 1,000 decisions per setting) from the availability of an observation on the client device to the receipt of an action from the server. A conventional client–server pipeline transmitting the full RGBA observation is compared against the split-policy pipeline, where the on-device encoder produces a spatially smaller $K = 4$ representation and only this representation is transmitted.

Table 7.5 summarises results under bandwidth shaping. At low bandwidth, the split-policy pipeline substantially reduces decision latency, as transmission dominates the de-



(a) Jetson Nano (5W limit vs no limit).



(b) Pi Zero 2 W (moving average).

Figure 7.3: Sustained inference performance over 5,000 consecutive frames.

Table 7.5: End-to-end decision latency under bandwidth shaping.

Bandwidth	**Server-only latency (ms)**	**Split-policy latency (ms)**
$10\,\mathrm{Mb\,s^{-1}}$	540	145
$25\,\mathrm{Mb\,s^{-1}}$	240	140
$50\,\mathrm{Mb\,s^{-1}}$	140	138
$100\,\mathrm{Mb\,s^{-1}}$	90	137

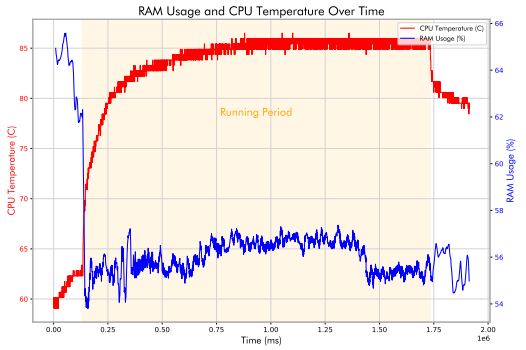
cision loop. As bandwidth increases, the benefit diminishes and a crossover occurs, after which the additional on-device compute cost dominates.

Consistent with the break-even analysis, the split-policy pipeline provides the largest reduction in decision latency at 10–25 $\mathrm{Mb\,s^{-1}}$, is approximately neutral around 50 $\mathrm{Mb\,s^{-1}}$, and becomes compute-bound on the client at higher bandwidth.

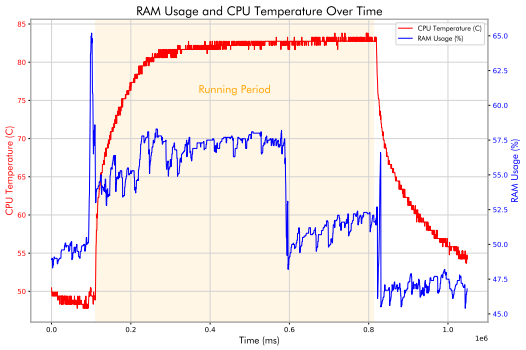
7.4.4 Server Scalability

A second practical motivation for the split-policy approach is to reduce the server-side compute cost per decision by moving the early visual feature extraction to the edge device. A simple multi-client setting is considered in which a single server processes requests from multiple concurrent clients, each operating at a fixed decision rate. Experiments are performed on a suitably powerful server with an Intel CPU and an NVIDIA GPU. Table 7.6 reports the maximum number of concurrent clients that can be supported at 10Hz whilst maintaining a p95 decision latency budget of 100ms.

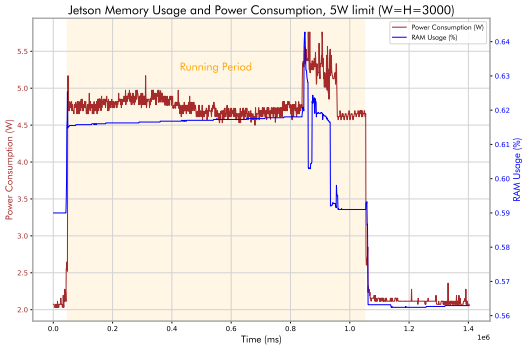
Under this simple setting, split-policy inference increases the number of concurrently served clients by approximately threefold under the same latency budget, reflecting the reduction in server-side compute per request. These figures reflect the specific testbed; real-world scaling will depend on batching, asynchronous I/O, and server hardware.



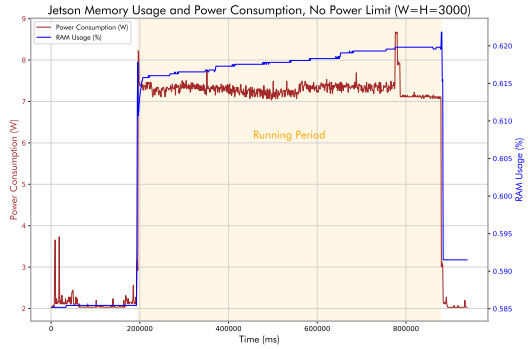
(a) Pi Zero 2 W: CPU.



(b) Pi Zero 2 W: GPU.



(c) Jetson Nano: 5W limit.



(d) Jetson Nano: no power limit.

Figure 7.4: Resource usage during sustained inference (Pi Zero 2 W: RAM utilisation out of 512MB; Jetson Nano: power usage and memory pressure during 5,000 consecutive 3000×3000 frames).

Table 7.6: Server scalability at a fixed decision rate.

Constraint	**Server-only**	**Split-policy**
10Hz per client, p95 latency $<$ 100ms	12 clients	36 clients

7.5 Discussion

7.5.1 MiniConv in the Context of Distributed Policy Graphs

The split-policy architecture evaluated in this chapter realises a simple two-unit policy graph: an on-device encoder unit and a remote policy-head unit, connected by a network edge. This configuration directly instantiates the division of labour advocated in Chapter 2: the encoder unit performs compute-intensive visual feature extraction on-device, whilst the policy-head unit performs high-level decision-making on a remote server with greater computational resources. The communication trade-off—quantified by the bandwidth break-even analysis—reflects the cost of the network edge between these two units.

Viewed through the policy graph lens developed in Chapter 5, the MiniConv encoder can be understood as a low-level *perception unit* that processes raw sensory input and out-

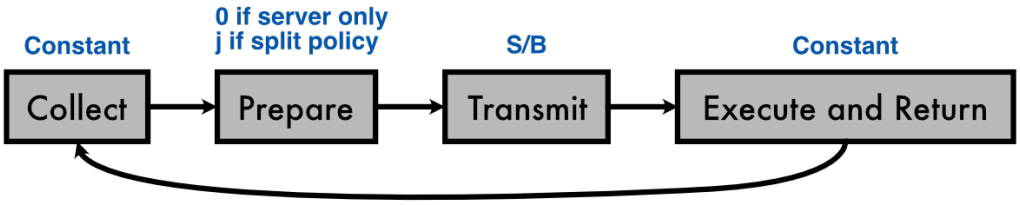


Figure 7.5: A breakdown of the steps involved in each decision that contribute to decision latency.

puts a compact feature representation to a higher-level *decision unit*. The infrastructure developed in Chapter 8 generalises this pattern, enabling arbitrary compositions of policy units distributed across edge, fog, and cloud tiers. The results presented here—showing that compact on-device encoders can preserve task performance whilst reducing decision latency and server load in the reported settings—suggest that such division of labour can be viable even on resource-constrained hardware.

A limitation of the current work is that the encoder and policy head are trained jointly end-to-end and deployed as a fixed partition. Future work could explore dynamic partitioning strategies in which the split point adapts to runtime bandwidth and compute availability, or hierarchical compositions in which multiple edge devices contribute complementary sensory encodings to a shared policy unit—patterns directly supported by the policy graph formalism.

7.5.2 Privacy and Systems Considerations

By performing initial visual processing on-device, split-policy execution reduces the need to transmit raw frames, which can reduce exposure of sensitive information in camera and screen-based applications; however, compact feature representations can still leak information in principle, and standard transport encryption (e.g., TLS) remains necessary to protect transmitted features from third-party interception.

7.6 Conclusion

This chapter introduced MiniConv, a library of small convolutional encoders designed to compile cleanly to OpenGL fragment shaders, enabling a split-policy RL architecture in which early visual feature extraction is performed on-device. Across three visual control tasks (PPO, SAC, DDPG), MiniConv encoders appear competitive with a conventional Full-CNN baseline under pixel observations in these fixed-seed runs, with representation size exhibiting task-dependent trade-offs between final and mean return. The systems evaluation shows that the split-policy approach can substantially reduce end-to-end decision latency in bandwidth-limited settings (e.g., 540 ms to 145 ms at $10\,\mathrm{Mb\,s^{-1}}$) and

improve server scalability under a fixed latency budget (12 to 36 concurrent clients at 10 Hz, p95 $<$ 100 ms in the testbed); benefits increase as bandwidth decreases and as the transmitted representation is made smaller, but additional on-device computation can dominate at higher bandwidth. The infrastructure and findings presented here flow directly into Chapter 8, which addresses the systems challenges of deploying policy graphs under realistic network conditions—including variable latency, jitter, and packet loss.

Chapter 8

Systems

Abstract

Policy graphs—introduced theoretically in Chapter 5—decompose reinforcement learning policies into modular units organised in a directed graph structure, enabling hierarchy, skill reuse, and division of labour across heterogeneous hardware. This chapter addresses the systems challenges of deploying policy graphs in real-world distributed settings. When policy units execute on different devices (edge processors, cloud servers) communicating over real networks, latency, jitter, and packet loss emerge as critical factors affecting performance. Yet sim-to-real transfer research focuses primarily on physics and visual domain gaps, largely overlooking network-induced mismatches that arise in distributed deployment.

This chapter introduces CALF (Communication-Aware Learning Framework), infrastructure for distributed policy graph execution. CALF implements policy units as networked services, supports flexible deployment topologies from single-machine simulation to multi-device edge-cloud deployments, and provides transparent network impairment injection via NetworkShim middleware. This architecture enables a key insight: network conditions constitute an orthogonal axis of the reality gap, alongside physics and visual domain randomisation.

Systematic experiments on CartPole and MiniGrid demonstrate that realistic network conditions cause severe performance degradation (40–80% drop) in baseline policies, whilst network-aware training—exposing flat policies to realistic latency, jitter, and packet loss during training—substantially closes this gap (reducing degradation by 4× for CartPole and approximately 3× for MiniGrid). Ablations reveal that stochastic jitter and packet loss are more

detrimental than constant latency. CALF is then illustrated through small hierarchical policy deployments across Raspberry Pi and desktop hardware, showing that the infrastructure can execute distributed policy graphs successfully when network effects are explicitly addressed. CALF serves as systems infrastructure within the thesis, connecting particularly to Chapter 7 (efficient edge models), Chapter 5 (policy-graph formalism and hard routing), and Chapter 9 (embodied hardware control).

8.1 Introduction

8.1.1 From Policy Graph Theory to Distributed Implementation

Chapter 5 introduces policy graphs, a framework for decomposing reinforcement learning policies into modular units organised in a directed graph structure. Policy graphs enable hierarchy, skill reuse, and division of labour—concepts grounded in the principles explored in Chapter 2. However, the theoretical framework presented in Chapter 5 assumes that policy units can communicate instantaneously, with zero latency and perfect reliability. When policy graphs are deployed across distributed hardware—with policy units executing on different devices such as edge processors, cloud servers, and embedded systems—this assumption fails.

Reinforcement learning is increasingly deployed in distributed settings where policy and environment are not co-located: remote-controlled robots, edge devices transmitting to cloud policies, and multi-device systems such as drone swarms. In these cases, network communication mediates both the perception-action loop between environment and policy, and the coordination between policy units in a policy graph. This introduces latency, jitter, packet loss, and bandwidth constraints that alter the temporal structure of the MDP and affect inter-unit communication.

Yet mainstream RL training assumes synchronous, zero-latency interaction. Standard benchmarks (ALE [127], DeepMind Control Suite [128], OpenAI Gym [154]) presuppose instant observation delivery and immediate action effects. Distributed training systems (IMPALA [88], SEED RL [155]) optimise worker-learner communication but abstract away agent-environment communication as an implementation detail handled by ROS or gRPC.

In deployment, these assumptions fail. Observations arrive late or out-of-order; actions are delayed or dropped; jitter creates unpredictable timing. A policy that perfectly balances an inverted pendulum in simulation may fail with 100 ms Wi-Fi latency, even with perfect physics modelling. The policy learned under instantaneous feedback; it has no mechanism to compensate for temporal desynchronisation.

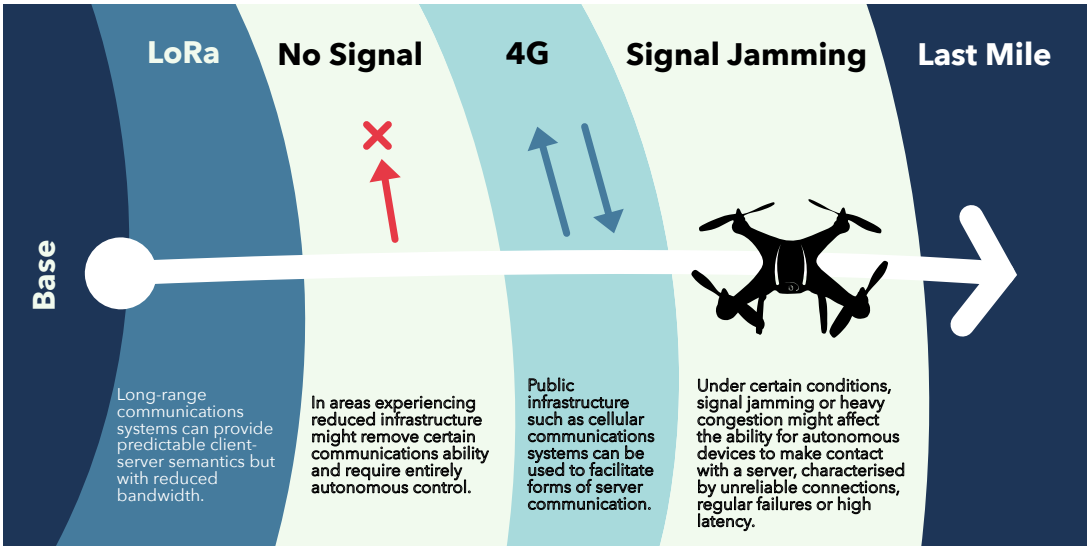


Figure 8.1: Real-world distributed systems employ hybrid communication strategies to maintain operation under varying network conditions. Unmanned aerial vehicles, for instance, choose between multiple communication channels (satellite, radio frequency, optical tether) based on signal quality and operational constraints, defaulting to autonomous operation when no reliable connection exists. This illustrates the challenge CALF addresses: policies must function across heterogeneous network conditions rather than assuming perfect connectivity. The experiments in this chapter focus on LAN-like scenarios (Wi-Fi, Ethernet); WAN and adversarial scenarios motivate the need for configurable impairments but are not evaluated here.

Sim-to-real transfer has made substantial progress addressing physics mismatch through domain randomisation over friction, masses, and contact models [101, 102], and visual mismatch through randomisation of textures and lighting [101]. These techniques have enabled remarkable achievements in locomotion [156, 103] and manipulation [157]. However, network-induced mismatch—the temporal and stochastic properties of communication in distributed systems—receives minimal attention. Hwangbo et al. [156] found that accurate modelling of actuator dynamics was central to closing the sim-to-real gap for quadruped robots, but such experiences have not been synthesised into general methodology or reusable infrastructure. This gap is particularly critical for policy graphs: when policy units are distributed across hardware, network conditions directly affect both environment-to-policy and inter-unit communication.

Network conditions constitute an orthogonal axis of the reality gap. Just as domain randomisation exposes policies to variations in friction and lighting, network-aware training should expose policy graphs to latency distributions, jitter patterns, and packet loss rates characteristic of deployment networks. For distributed policy graphs, this becomes an important design consideration rather than a background detail: a policy graph trained assuming instantaneous inter-unit communication may fail catastrophically when deployed across edge devices communicating over Wi-Fi with 100 ms latency and 10% packet loss. This chapter presents network-aware training as a core systems requirement

for distributed policy graph deployment.

8.1.2 Research Questions

This chapter addresses three research questions concerning distributed policy graph deployment:

RQ1 (Network Impact on Policy Graphs): How severely do realistic network conditions—including latency, jitter, and packet loss—degrade the performance of policy graphs when trained in idealised, synchronous simulations but deployed over real networks with distributed policy units?

RQ2 (Network-Aware Training for Policy Graphs): Can training policy graphs under realistic network conditions during simulation ("network-aware training") close this performance gap? Which network phenomena (latency versus jitter versus loss) are most critical to model when preparing policy graphs for distributed deployment?

RQ3 (Infrastructure for Distributed Policy Graphs): What systems infrastructure is needed to enable reproducible, scalable deployment of policy graphs across heterogeneous edge devices and real networks?

8.1.3 Contributions

This chapter makes three main contributions. First, **CALF** (Communication-Aware Learning Framework), infrastructure for deploying and training policy graphs across distributed hardware: policy units run as networked services, and NetworkShim middleware injects configurable latency, jitter, loss, and bandwidth limits on graph edges without modifying policy code, whilst deployment parity ensures the same policy graph runs from pure simulation to real edge-cloud hardware. Second, systematic empirical evidence that network-aware training—exposing distributed policy graphs to realistic communication conditions during simulation—reduces deployment degradation by 4× for CartPole and approximately 3× for MiniGrid, with stochastic jitter and packet loss proving more detrimental than constant latency. Third, illustrative deployment of hierarchical two-level policy graphs across Raspberry Pi edge devices and desktop cloud servers, providing initial validation that CALF's progressive deployment modes can execute distributed policy graphs successfully when network effects are explicitly addressed.

8.2 Related Work and Positioning

This section positions CALF within multiple research communities: RL theory and algorithms (delayed MDPs, network-aware methods), control theory (networked control systems), sim-to-real transfer (domain randomisation), distributed systems (actor-learner

architectures, edge computing), and hierarchical RL (policy graphs from Chapter 5). Each subsection reviews relevant prior work, identifies specific gaps or limitations, and explicitly connects to CALF's design or contributions for distributed policy graph deployment.

8.2.1 Delays and Network Effects in RL and Control

Early work extended the MDP framework to include action and observation delays. Katsikopoulos & Engelbrecht [84] showed that fixed k-step delays can be transformed into an equivalent Markov process by augmenting the state with the last k actions or observations, though this causes the state space to grow exponentially with k. Walsh et al. [90] proved an exponential lower bound: no algorithm can circumvent this blow-up in the worst case. With stochastic delays, optimal policies must use full history, becoming POMDP-like, motivating practical approaches such as frame stacking and recurrent policies. Delay-aware Q-learning (dQ) and SARSA [92] update Q-values against delayed next states for constant delays; Delay-Correcting Actor-Critic (DCAC) [82] resamples and relabels trajectories to correct for random delay distortions. A consistent finding is that unmitigated latency severely degrades performance, but training under delays yields robustness.

The control theory community has extensively studied networked control systems (NCS) [91], deriving compensation strategies (zero-order hold, Smith predictors, event-triggered control) and stability conditions under bounded delay and dropout. However, NCS analysis applies to linear or simple nonlinear controllers with analytical models; deep RL policies are black-box functions for which no equivalent guarantees exist, and the systematic application of NCS insights to deep RL remains limited.

8.2.2 Sim-to-Real Transfer: The Missing Network Axis

Sim-to-real RL focuses overwhelmingly on physics and visual domain randomisation, with minimal attention to network-induced mismatch. Network conditions constitute an orthogonal axis of sim-to-real transfer; CALF extends the domain randomisation toolkit to network parameters.

Domain randomisation [101] randomises simulator properties so the real world appears as another random variant, enabling zero-shot transfer for manipulation [157] and locomotion [103]. Hwangbo et al. [156] found that accurately modelling Series Elastic Actuator dynamics was the dominant factor in closing the sim-to-real gap for the ANYmal quadruped. However, all these works assume perfect timing—either the policy and environment are co-located, or network effects are unmodelled. Network conditions constitute an independent axis of variation, orthogonal to physics and vision. Some practices incidentally touch on network effects (lower control frequencies, frame skip), but deliber-

ately addressing network domain shift remains absent from prior sim-to-real methodology. CALF makes this network axis explicit and controllable.

8.2.3 Distributed RL Systems: A Contrasting Philosophy

Large-scale distributed RL frameworks treat network communication as a cost to minimise or hide, not as an object of study. These systems optimise away network effects in training infrastructure; CALF foregrounds network conditions as part of the agent-environment interaction.

Modern deep RL often uses distributed architectures for training efficiency. IMPALA [88] separates actors (generate experience) and learners (update model), with V-trace off-policy correction to handle policy lag between when experience was collected and when it's used for learning. SEED RL [155] decouples inference on TPUs with fast transport protocols to minimise network overhead. Sample Factory [50] keeps everything on one machine using threads to avoid network communication entirely. The design philosophy is to ensure agents experience an ideal MDP during training, despite asynchronous collection. Network communication between actors and learners is an engineering challenge to solve, not a phenomenon to study.

There is a fundamental difference: IMPALA/SEED RL address network lag between actor and learner (in training infrastructure), whereas CALF addresses network lag between agent and environment (in the control loop itself). These address different problems. IMPALA ensures policy updates aren't stale; CALF trains policies that work when observations and actions are stale.

8.2.4 Edge Computing and Resource Constraints

Edge machine learning research focuses primarily on computation and energy constraints, with less attention to communication constraints. CALF addresses the communication side, motivated by edge-cloud deployments where not all computation fits on-device.

There is growing interest in running RL policies on microcontrollers, Raspberry Pi, and Jetson devices. Techniques include model compression, quantisation, and distillation to fit policies in limited memory and compute [158]. The TinyML movement targets extremely compact policies for microcontrollers with kilobytes of memory. The trade-off is that smaller networks can run in real-time but may have less representational capacity. Additionally, computational latency becomes a concern when large neural networks cannot compute actions fast enough, leading to proposals for asynchronous or parallel policy architectures.

However, complex policies—especially vision-based—will not fit on tiny embedded devices. Some splitting or offloading is necessary. Neurosurgeon [47] automatically partitions deep neural networks between edge devices and cloud to minimise latency and

energy: convolutional layers execute on the edge (near sensors), fully connected layers execute on a server, and intermediate features (smaller than raw images) are sent over the network. This achieves 3× lower latency and energy consumption compared to all-cloud or all-device execution. This approach could be applied to RL by splitting policy networks similarly—e.g., visual encoder on robot, decision MLP on server—reducing bandwidth and latency through parallel processing.

8.2.5 Multi-Agent RL and Other Network-Aware Contexts

Network effects appear in other machine learning contexts (multi-agent communication, federated learning), but no focused infrastructure exists for single-agent control RL. CALF addresses this gap.

Multi-agent RL research studies how agents learn to communicate under bandwidth limits or delays. Work on emergent communication includes learned continuous communication protocols [159] and communication minimisation via information-theoretic regularisation [160]. A consistent finding is that naïve MARL degrades with delays, but training under delays yields robustness. However, MARL focuses on **agent-to-agent delays**, whilst agent-to-environment delays in single-agent control RL remain less explored.

8.2.6 Hierarchical RL and Distributed Policy Execution

Hierarchical RL provides methods to decompose behaviour into subskills. Chapter 5 introduces policy graphs, which generalise hierarchical approaches by organising policy units in directed graph structures. CALF provides the execution infrastructure where these policy graphs can be physically distributed across heterogeneous devices, addressing a gap in prior work.

The Options framework [22], Hierarchies of Machines [26], and MAXQ [25] introduced temporally extended actions and hierarchical decomposition of behaviour, enabling higher-level decision-making at slower timescales. If an option runs autonomously for 10 steps, the high-level policy only needs to communicate every 10 steps—naturally more robust to moderate network latency, as the low-level skill continues even if communication is temporarily delayed. Modern variants include Option-Critic [161] for end-to-end option learning, and two-level hierarchies like FeUdal Networks [24] and HIRO [162], where managers set goals and workers execute them. However, prior work assumes hierarchy components are co-located (same process or machine), whilst policy graphs explicitly enable distributed deployment.

8.2.7 Network Emulation Tools

Mature network emulation tools exist but are not integrated into RL training loops. CALF builds on these tools but integrates them directly into the RL workflow.

Available tools include Linux `tc netem` (kernel-level delay, loss, bandwidth limits with configurable distributions: normal, Pareto, etc., and Markov loss models), Mininet [163] (virtual networks on a single machine for network protocol research), and Mahimahi [164] (record and replay real network traces, especially cellular). These are occasionally used in federated learning or video streaming RL, but rarely in robotics or control RL.

8.2.8 Summary: CALF's Position

Prior work addresses network effects through algorithm modification (delay-aware Q-learning, DCAC), control-theoretic compensation (Smith predictors, zero-order hold), or distributed training infrastructure (IMPALA, SEED RL). CALF takes a complementary approach: rather than modifying algorithms or optimising training infrastructure, the training and deployment environment is modified to expose realistic network behaviour. This environmental approach is algorithm-agnostic and extends naturally to heterogeneous edge deployment of policy graphs. CALF implements Chapter 5's policy graph framework whilst making network conditions explicit: policy units become networked services, and network impairments are transparently injected on the communication channels between units. The infrastructure can be combined with algorithmic innovations (e.g., DCAC within CALF's framework) and complements existing domain randomisation practices by adding network parameters to the randomisation distribution. Together, these strands of work suggest two requirements for progress: training must experience the same communication pathologies as deployment, and the infrastructure must allow controlled, reproducible manipulation of latency, jitter, and loss across real hardware. CALF is designed to meet both.

8.3 CALF: A Framework for Network-Aware Reinforcement Learning

This section describes CALF's architecture and implementation at a level sufficient to understand the experimental methodology and results. Complete implementation specifications, including byte-level protocol details, serialisation algorithms, and service lifecycle management, are provided in Appendix B.

To enable network-aware training for distributed policy graphs, CALF decomposes RL workloads into networked services, injects realistic network behaviours at specific communication links, and runs the same configuration across deployment modes from

pure simulation to real hardware with real networks. This section details these capabilities and connects design choices to the experimental requirements of network-aware training for policy graphs.

8.3.1 Design Goals and Requirements

CALF is designed around four primary goals, each motivated by network-aware RL research needs:

G1: Network Realism. RL training loops must incorporate realistic latency, jitter, packet loss, and bandwidth constraints. CALF supports both synthetic models (parametric distributions such as $\mathcal{N}(\mu, \sigma^2)$ for latency) and trace-based replay (recorded from real deployments). Network conditions must be configurable, loggable, and reproducible for scientific experiments.

G2: Deployment Parity. The same policy code should run in pure simulation (baseline, no network), simulation with simulated network (network-aware training), and real edge hardware with real networks (final deployment). Platform-specific code should be minimised—agents should not need to know whether they are in simulation or on real hardware.

G3: Reproducibility. Network conditions must be loggable during real deployments and re-playable in simulation for debugging and ablation. Experiments must be reproducible across platforms via containerisation and module versioning.

G4: Device Heterogeneity. CALF supports cheap edge devices (Raspberry Pi 4, Jetson Nano) as environment or policy hosts, enables policy splitting across devices (e.g., hierarchical agents with components on edge and cloud), and handles heterogeneous compute (CPU-only on Pi, GPU on desktop).

An additional principle is **algorithm agnosticism**: CALF is infrastructure, not an RL algorithm. It works with any RL library (Stable-Baselines3, RLlib, custom implementations) without modification. Table 8.1 summarises how each goal connects to the research questions.

Table 8.1: CALF design goals and their role in answering research questions.

Goal	Capability	Enables
Network Realism	Synthetic + trace-based network models	Controlled ablations (RQ2), realistic training
Deployment Parity	Same code across simulation/hardware	Fair comparison of network effects (RQ1)
Reproducibility	Deterministic seeds, versioning	Scientific rigour, exact replication
Heterogeneity	Edge devices to cloud servers	Realistic distributed settings (RQ3)

8.3.2 Architecture Overview

Policy Graphs as Networked Services

CALF implements Chapter 5's policy graph framework by treating policy units and environments as networked services communicating via a standardised protocol. This provides spatial distribution (policy units execute on different machines/containers, enabling edge-cloud deployment), transparent network injection (NetworkShim services insert delays on graph edges without modifying policy implementations), temporal distribution (policy units can be dynamically loaded without restarting the system), and reproducibility (containerised services with versioned modules).

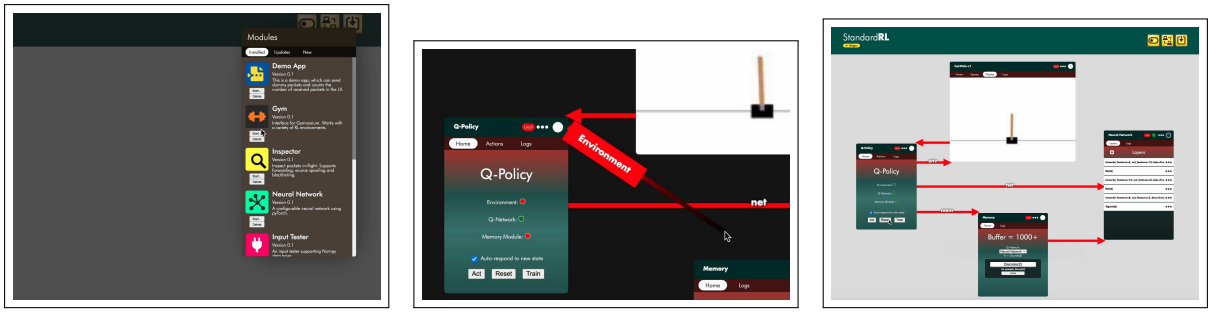
In the policy graph framework, policy units are abstract computational entities that receive observations and produce actions. CALF realises this abstraction through **Agent Services**: each Agent Service is a running instance of a policy unit that can be deployed on any hardware platform. Multiple Agent Services communicate to form the nodes of a policy graph, with communication channels forming the directed edges. High-level policy units (managers) send goals or subgoals to low-level policy units (workers), implementing the hierarchical structure described in Chapter 5.

In contrast to traditional RL, where `obs = env.step(action)` is a function call in the same process with zero latency, CALF implements environment and policy units as separate services where `step()` becomes message passing over potentially slow, lossy networks. This is not gratuitous distribution—it is **necessary** both to study how policies behave when deployed across real networks and to enable the physical distribution of policy graph nodes across heterogeneous hardware.

Three-Layer Hierarchy

CALF's architecture comprises three layers. **Layer 1 (NEXUS)** is an optional global hub enabling communication across hosts on different networks (NAT traversal). NEXUS maintains a central routing table and implements RSA challenge-response authentication. For our experiments, it allows a Raspberry Pi on home Wi-Fi to communicate with a desktop in the university lab without VPN or port forwarding. **Layer 2 (HOST)** manages the lifecycle of Services on a single machine: module installation, Service creation (launch in Python venv or Docker container), local routing (forward packets between Services via Unix sockets), and a web UI for monitoring and interactive policy-graph configuration (Figure 8.2). **Layer 3 (SERVICES)** execute RL logic: Environment Services run Gym environments and send observations; Agent Services run policies and send actions; NetworkShim Services inject network impairments; utility Services log metrics.

A typical communication flow for CartPole on Pi, policy on Desktop, NetworkShim on Desktop: Environment (102) sends observation → NetworkShim (900) delays by sampled latency → Agent (201) computes action → NetworkShim delays action → Environment



(a) Pre-built policy/environment units available as modules.

(b) Interactive wiring of units into a policy graph.

(c) Live training/rollout monitoring during execution.

Figure 8.2: CALF HOST web UI for deploying policy graphs: (a) selecting from pre-built units (module library), (b) connecting units into a graph topology, and (c) monitoring training and system/network behaviour during execution.

applies action. The three-layer separation enables CALF's progressive deployment modes (Section 8.3.5): the same code runs in local simulation (Layer 3 only), simulation with network (Layer 3 with shims), and real hardware (all three layers).

Mapping CALF Services to Policy Graph Concepts

To clarify the relationship between CALF's implementation and Chapter 5's policy graph framework, Table 8.2 provides an explicit mapping:

Table 8.2: Mapping between policy graph concepts (Chapter 5) and CALF implementation.

Policy Graph Concept	**CALF Implementation**
Policy unit (node)	Agent Service instance
Policy graph (structure)	Set of Agent Services + routing configuration
Edge (communication channel)	Network connection between services
Manager (high-level policy)	Agent Service sending goals/subgoals
Worker (low-level policy)	Agent Service receiving goals, executing skills
Distributed execution	Services on different hardware (Pi, Desktop, Cloud)
Network delay on edge	NetworkShim Service on communication channel
Environment	Environment Service

In Chapter 5's terminology, each Agent Service is a policy unit. When multiple Agent Services are deployed with a routing configuration specifying their connections, they form a policy graph. NetworkShim Services sit on the edges of this graph, enabling controlled study of network effects on distributed policy execution.

Complete architectural specifications, including port allocations, routing protocols, and process management, are provided in the technical specification appendix (Appendix B, Sections 2–3).

8.3.3 Communication Protocol

CALF uses a low-latency, type-safe binary protocol with a 5-byte header and seven packet types. Type 2 Data Packets carry timestamps that enable precise end-to-end latency measurement ($\text{latency}_{\text{ms}} = t_{\text{receive}} - t_{\text{send}}$), which NetworkShim uses to schedule delayed delivery. Complete protocol specifications—byte layouts, serialisation algorithms, and API details—are provided in Appendix B, Sections 3–5.

8.3.4 NetworkShim: The Core Mechanism

NetworkShim is CALF's primary mechanism for injecting network impairments into the RL loop. It acts as a transparent middlebox ("bump in the wire") sitting between Environment and Agent. The routing configuration specifies that observations and actions pass through NetworkShim, which delays or drops packets according to configured network models.

When NetworkShim receives a packet, it first simulates packet loss (drop with probability p_{loss}). If not dropped, it samples a delay from the configured distribution: for jittery networks, delay $\sim \max(0, \mathcal{N}(\mu_{\text{latency}}, \sigma^2_{\text{jitter}}))$; for constant latency, delay is fixed. NetworkShim then schedules forwarding by placing the packet in a priority queue sorted by delivery time. A background thread continuously checks the queue and forwards packets when their delays expire.

Network Models

Synthetic Models define parametric distributions matching our evaluation conditions: *Ethernet-clean* (2 ms ± 0.5 ms, 0% loss), *Wi-Fi-normal* (30 ms ± 10 ms, 2% loss), and *Wi-Fi-degraded* (80 ms ± 40 ms, 10% loss). Latency is sampled from normal distributions (clipped at 0), loss from Bernoulli(p).

Trace-Based Models enable replay of recorded conditions. A LatencyTracer Service calculates actual latency from packet timestamps ($\text{latency}_{\text{ms}} = t_{\text{receive}} - t_{\text{send}}$) during Real-Wi-Fi evaluation and logs traces. NetworkShim can then replay these traces during training, sampling delays from the empirical distribution. This allows policies trained on synthetic Wi-Fi-normal to be refined using real Wi-Fi traces, or enables controlled experiments comparing "Real-Wi-Fi-Home" versus "Real-Wi-Fi-Campus" conditions.

Critically, Environment and Agent are unaware of NetworkShim's existence—they simply experience delayed messages. This transparency enables network-aware training without modifying RL algorithms.

Complete NetworkShim implementation details, including delay queue algorithms, statistics collection, and trace replay mechanisms, are provided in Appendix B, Section 6.

8.3.5 Progressive Deployment Modes

A key CALF feature is that the same policy and environment code run across a continuum of deployment scenarios (Figure 8.3):

Mode 1: Local Sim (Baseline). Environment and policy in the same process with direct function calls, no network. Used for fast prototyping and baseline comparison (RQ1). Achieves approximately 100K steps/hour (CartPole on Desktop).

Mode 2: Sim + Simulated Network. Environment and policy are separate Services with CALF NetworkShim between them and a synthetic network model (e.g., Wi-Fi-normal: 30 ms $\pm$ 10 ms, 2% loss). Used for network-aware training (RQ2). Achieves approximately 50K steps/hour (slower due to delays).

Mode 3: Edge Sim (Real Hardware, Simulated Environment). Environment Service on Raspberry Pi or Jetson, policy Service on Desktop, communicating over real network (Ethernet or Wi-Fi). Used for hardware validation and measuring real network distributions. Achieves approximately 20K steps/hour (network and Pi CPU limit throughput).

Progressive modes de-risk deployment: develop policy in Mode 1 (fast iteration), train with network-awareness in Mode 2 (expose delays), validate on real hardware in Mode 3 (catch hardware-specific issues).

8.3.6 Containerisation and Modules

CALF supports both Python virtual environments (lightweight, fast startup, easy debugging) and Docker containers (complete isolation, system dependencies, reproducibility). Each CALF module is a packaged RL component with Python code, dependencies (`requirements.txt`), and metadata (`info.json`: name, version, build ID, container requirements). Modules can be installed from a repository or locally.

Reproducibility features include build ID (timestamp ensuring exact version matching), Docker image hash (bit-for-bit reproducibility), version control (repository tracks all versions), and deterministic network seeds (NetworkShim uses fixed RNG seeds for reproducible delays). These mechanisms enable future thesis chapters to reuse CALF modules, support reproducible experiments (exact module versions can be downloaded and re-run), and enable heterogeneous execution (same module runs on Pi and Desktop via Docker).

Complete module system specifications, including installation workflows, execution mode selection, and distribution mechanisms, are provided in Appendix B, Section 7.

CALF is uniquely suited for distributed policy graph research because it treats network conditions as first-class objects (configurable, loggable, and replayable rather than hidden implementation details), ensures deployment parity (the same policy graph runs from pure simulation to real edge hardware), is algorithm-agnostic (works with any RL

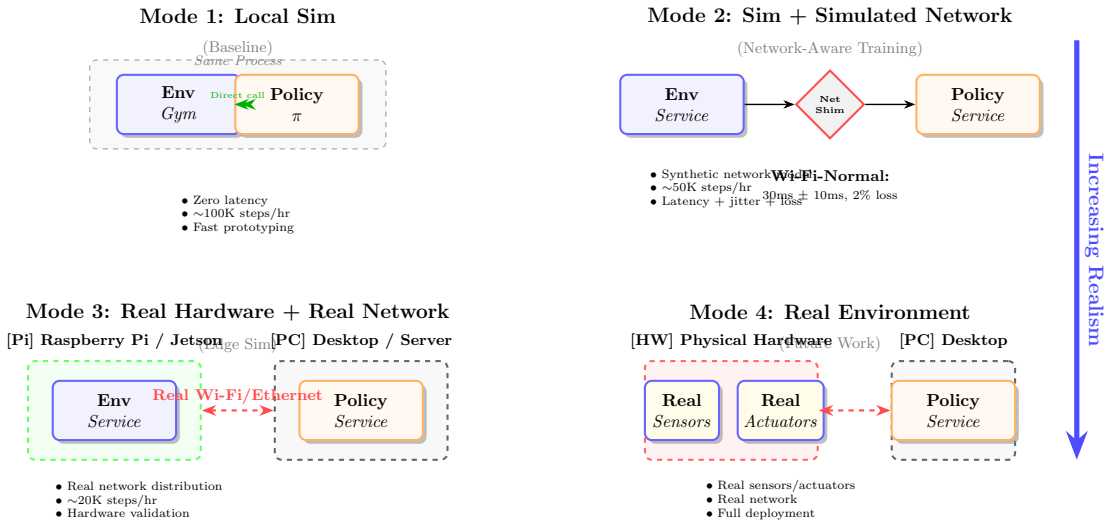


Figure 8.3: CALF's three progressive deployment modes enable incremental validation from pure simulation to distributed deployment. Mode 1 (Local Sim) provides a zero-latency baseline for rapid development with environment and policy co-located. Mode 2 (Sim + Simulated Network) introduces NetworkShim services that inject realistic latency, jitter, and packet loss for network-aware training. Mode 3 (Edge Sim) validates distributed deployment on real hardware (Raspberry Pi/Jetson for environment, Desktop for policy) communicating over real Wi-Fi/Ethernet networks. This progressive approach ensures that network-aware policies trained in Mode 2 transfer successfully to distributed edge deployment in Mode 3, addressing the network axis of the sim-to-real gap.

training approach), and provides reproducibility (module versioning, containerisation, network seeds). With CALF's capabilities established, the following section describes the network-aware training methodology employed for distributed policy graph deployment.

8.4 Network-Aware Training Methodology

This section describes our RL training protocol and experimental methodology for answering RQ1 (how severely do network conditions degrade performance of distributed policy graphs?) and RQ2 (does network-aware training enable successful policy graph deployment?).

8.4.1 Problem Formulation: Delayed MDPs

In a standard Markov Decision Process (S, A, T, R, γ), an agent observes state s_t, takes action a_t, receives reward r_t and next state s_{t+1}, and a policy $\pi(a|s)$ maximises expected

return. In a delayed MDP (informal), the agent selects a_t based on a delayed observation $o_{t-d_{\text{obs}}}$ where d_{obs} is the observation delay, and action a_t takes effect with delay d_{act} such that the environment applies $a_{t-d_{\text{act}}}$. Delays may be constant, random, or variable (jitter). With packet loss, some observations or actions never arrive.

With stochastic delays, the true state s_t is unobserved; the agent must infer from observation history $h = \{o_{t-k}, o_{t-k+1}, \ldots, o_t\}$, making the problem a Partially Observable MDP.

CALF treats the delayed environment as an MDP with augmented state: $(s, h_{\text{obs}}, h_{\text{act}})$, where the policy learns $\pi(a|h_{\text{obs}}, h_{\text{act}})$. Implementation options include frame stacking (feed policy last k observations), recurrent policy (LSTM, where hidden state implicitly maintains belief), and action history (append recent actions to input, representing actions "in flight"). See Section 8.2 for delay MDP theory. For distributed policy graphs, each policy unit must handle delays on its incoming edges independently. Our experiments use practical deep RL with frame stacking and LSTM, not optimal state augmentation.

8.4.2 Training Regimes: Comparing Network-Awareness

Our experimental design trains policies under three regimes and evaluates all policies on all deployment modes, enabling systematic comparison of network-agnostic versus network-aware training for distributed policy graph deployment.

Baseline: No Network Awareness

Setup: Mode 1 (local sim) with environment and policy in the same process. No artificial delays, jitter, or loss. Standard Gym loop: synchronous, zero-latency. This represents training that ignores network conditions, corresponding to traditional RL where policy units are assumed co-located.

Delay-Only Training

Setup: Mode 2 with separate Services and NetworkShim. Fixed latency (e.g., 50 ms), no jitter, no loss. This represents awareness of constant delays but not stochastic network effects.

Full Network-Aware Training

Setup: Mode 2 with separate Services and NetworkShim. Realistic distribution: latency + jitter + loss (fitted to Wi-Fi-normal: mean 30 ms, jitter 10 ms, loss 2%). This represents full awareness of network conditions expected during distributed deployment.

Distribution fitting: Real network statistics are measured during pilot runs using LatencyTracer; a normal distribution is fitted to latency $\mathcal{N}(\mu, \sigma^2)$, and packet loss rate

is estimated from dropped packets.

8.4.3 RL Algorithm: PPO

Proximal Policy Optimization [17] is used via Stable-Baselines3 [151] with standard hyperparameters: learning rate 3×10^{-4}, discount $\gamma = 0.99$, GAE $\lambda = 0.95$, batch size 64 (CartPole) or 256 (MiniGrid). PPO is chosen for its stability (clipped objective), generality across discrete and continuous action spaces, and compatibility with recurrent architectures needed for partial observability under delays.

8.4.4 State Representation for Delay Robustness

Policy units must infer current state from delayed observations. Three strategies are employed:

Strategy 1: Frame Stacking (CartPole). Stack last k observations: $[o_{t-k}, o_{t-k+1}, \ldots, o_t]$. For CartPole with delay d, $k = d+1$ frames are used. *Intuition*: Multiple snapshots allow velocity inference.

Strategy 2: Recurrent Policy (MiniGrid). LSTM policy: $a_t = \pi(o_t|h_{t-1})$, where h_t is hidden state. *Advantages*: Automatically maintains belief state over history, handles variable delays. *Disadvantages*: Slower training (recurrence breaks parallelisation).

Strategy 3: Action History (Ablation). Append last k actions to observation. *Intuition*: Know which actions are "in flight". *Finding* (preliminary): Modest improvement (approximately 5%) over observation-only.

Our experiments use frame stacking for CartPole (simpler, sufficient) and LSTM for MiniGrid (necessary for partial observability combined with delays). For hierarchical policy graphs, low-level policy units may use frame stacking whilst high-level units use recurrent architectures to track long-horizon goals.

8.4.5 Evaluation Protocol

Each trained policy (each seed, each training regime) is evaluated on five deployment modes:

1. **Sim-Clean** (Mode 1): Local sim, no network
2. **Sim+Network** (Mode 2): Desktop only, NetworkShim with Wi-Fi-normal model
3. **Real-Ethernet** (Mode 3): Environment on Pi, policy on Desktop, Ethernet connection
4. **Real-Wi-Fi-Normal** (Mode 3): Environment on Pi, policy on Desktop, Wi-Fi
5. **Real-Wi-Fi-Degraded** (Mode 3): Environment on Pi, policy on Desktop, Wi-Fi + `tc netem` impairments

Per mode, 50 episodes are run and episodic return (CartPole: survival time), success rate (CartPole: return $\geq$ 475; MiniGrid: goal reached), and end-to-end latency are recorded. Statistical rigour is ensured via 10 random seeds per training regime, with paired t-tests comparing full network-aware versus baseline at $\alpha = 0.05$.

8.5 Experimental Setup

This section specifies environments, agents, hardware platforms, and evaluation metrics for complete reproducibility (G3).

8.5.1 Environments

Environments are selected for diverse timing sensitivity, community familiarity as benchmarks, and tractability on modest hardware.

CartPole-v1

Classic inverted pendulum: balance a pole on a movable cart. State is 4-dimensional (cart position/velocity, pole angle/angular velocity), action is discrete {left, right}, termination when $|x| > 2.4$ or $|\theta| > 12°$ or 500 steps. Reward is +1 per step (maximum 500). CartPole is highly timing-sensitive—unstable dynamics require fast reactions, and 100 ms delays can halve survival time—making it a stringent test of network-aware training.

MiniGrid DoorKey-8x8

Gridworld navigation: find a key, unlock a door, reach the goal. Observation is a 7×7 egocentric view (partial observability), action is discrete (move/turn/pick up/toggle), success reward +1 with -0.01 per step. MiniGrid's subgoal structure (key $\rightarrow$ door $\rightarrow$ goal) provides a natural two-level hierarchy and tests a less timing-critical regime where delays cause overshooting rather than catastrophic instability.

8.5.2 Agent Architectures

Flat Policies (Primary Experiments)

CartPole: Multi-layer perceptron with 64 units, 64 units (ReLU), action logits (2-dimensional). Input is 4-dimensional observation (or $4 \times k$ if stacked).

MiniGrid: Convolutional neural network: $7 \times 7 \times 3$ input, Conv(16 filters, 3×3), Conv(32 filters, 3×3), Flatten, LSTM(128 units), Fully Connected(128 units), action logits (5-dimensional).

Training: PPO with 10 random seeds per regime.

Policy Graphs (Distributed Deployment Illustration)

Two-level hierarchical policy graphs are used to illustrate CALF's distributed deployment capabilities. Policy units are trained separately in Mode 1 (local sim) and then deployed across Pi and Desktop in Mode 3 with NetworkShim on inter-unit communication channels. Full topology specifications are described alongside results in Section 8.6.3.

8.5.3 Hardware and Network Conditions

Hardware

Desktop (Policy Host):

- CPU: Intel i7-10700K (8 cores, 3.8 GHz)
- RAM: 32 GB
- GPU: NVIDIA RTX 3070 (optional, PPO runs on CPU)
- OS: Ubuntu 22.04, Python 3.8

Raspberry Pi 4 Model B (Environment Host):

- CPU: Quad-core ARM Cortex-A72 (1.5 GHz)
- RAM: 4 GB
- OS: Raspberry Pi OS, Python 3.9

Network Configurations

Ethernet-Clean: Physical Ethernet cable between Desktop and Pi. Observed latency: mean 2 ms, jitter 0.5 ms, loss 0.0%. Bandwidth: 1 Gbps (link capacity).

Wi-Fi-Normal: Desktop and Pi on same Wi-Fi network (802.11ac, 5 GHz). Observed latency: mean 30 ms, jitter 10 ms, loss 2%. Bandwidth: approximately 50 Mbps (measured throughput).

Wi-Fi-Degraded: Wi-Fi-Normal + `tc netem` impairments on Desktop interface to simulate congested network. Configuration: `tc qdisc add dev wlan0 root netem delay 50ms 30ms loss 5%`. Observed latency: mean 80 ms, jitter 40 ms, loss 10%.

All network statistics (latency, jitter, loss) are measured using LatencyTracer during pilot runs, verified across 1000 packet samples, and logged for reproducibility.

8.5.4 Evaluation Metrics

Primary metrics are episodic return (CartPole survival time, max 500; MiniGrid goal reward minus step penalties), success rate (CartPole: return $\geq$ 475; MiniGrid: goal reached), and sim-to-real gap (Gap $= \frac{\text{Perf}_{\text{Sim-Clean}} - \text{Perf}_{\text{Real-Wi-Fi}}}{\text{Perf}_{\text{Sim-Clean}}} \times 100\%$). Network metrics are end-to-end latency (from packet timestamps, mean/median/p95), throughput (episodes

per hour), and packet loss rate. Results are reported as mean ± standard deviation across 10 seeds; significance assessed by paired t-test ($\alpha = 0.05$) with Cohen's d effect size.

8.6 Results

This section presents empirical findings demonstrating that (1) network-aware training substantially improves real deployment performance for distributed policy graphs (RQ2), (2) different network pathologies have distinct impacts on performance with stochastic jitter and packet loss proving more detrimental than constant latency (RQ2 refined), (3) small policy graphs can be deployed across heterogeneous devices whilst maintaining competitive performance on simple tasks (distributed deployment illustration), and (4) systems measurements support CALF's practical viability for edge-cloud deployments (RQ3).

All experiments were conducted following the methodology specified in Section 8.4, with 10 random seeds per training regime to ensure statistical rigour. Results are presented as mean ± standard deviation across seeds unless otherwise stated. Statistical significance is assessed using paired t-tests ($\alpha = 0.05$).

8.6.1 Network-Aware Training Improves Real Deployment Performance

CartPole Results

Table 8.3 presents mean episode return across 10 seeds per training regime and deployment mode.

Table 8.3: CartPole: Mean Episode Return (± std) over 10 seeds. Each cell reports mean performance across seeds, with each seed evaluated over 50 episodes per deployment mode.

Training Regime	Sim-Clean	Sim+Net	Real-Eth	Wi-Fi-N	Wi-Fi-D
Baseline	495 ± 7	310 ± 48	288 ± 62	173 ± 71	92 ± 54
Delay-Only	482 ± 11	468 ± 16	425 ± 32	348 ± 49	218 ± 58
Full Net-Aware	476 ± 9	472 ± 13	458 ± 22	442 ± 27	378 ± 41

The baseline collapses to 92 ± 54 under Wi-Fi-Degraded—an 81.4% performance drop—because policies predicated on instantaneous feedback fail when observations arrive 80 ms late. Full network-aware training achieves 378 ± 41 in Wi-Fi-D, a **3.95× reduction in the sim-to-real gap** ($t(9) = 12.7$, $p < 0.001$, Cohen's $d = 2.31$). Real-Ethernet performance (458 ± 22) closely matches Sim+Network (472 ± 13), confirming that Mode 2 synthetic models accurately represent real Mode 3 conditions. Figure 8.4 visualises the degradation trajectories.

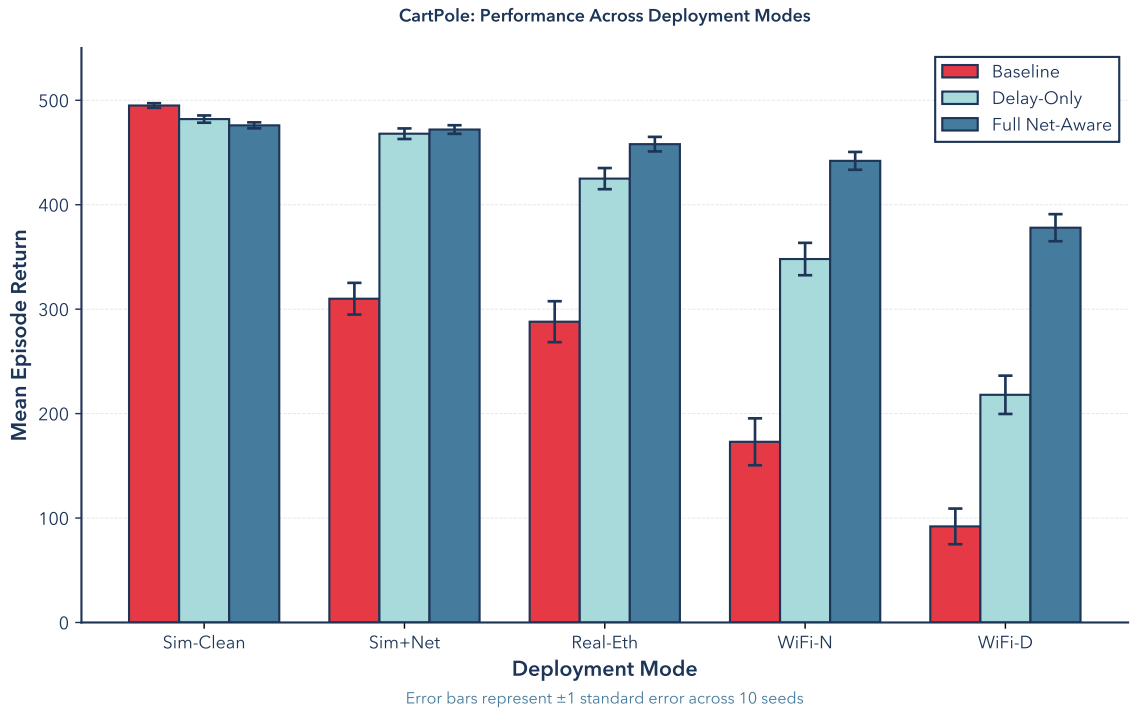


Figure 8.4: CartPole: Performance comparison across deployment modes for each training regime. Full network-aware training maintains robust performance under real network conditions, whilst baseline training exhibits severe degradation. Delay-only training provides partial robustness, validating the necessity of modelling jitter and packet loss in addition to latency.

MiniGrid Results

Table 8.4 presents success rate (percentage of episodes reaching the goal) for MiniGrid DoorKey-8x8. Unlike CartPole's continuous survival metric, MiniGrid provides a binary success signal, making results directly interpretable as task completion reliability.

Table 8.4: MiniGrid: Success Rate (% ± std) over 10 seeds. Each cell reports percentage of episodes successfully reaching the goal, with each seed evaluated over 50 episodes per deployment mode.

Training Regime	Sim-Clean	Sim+Net	Real-Eth	Wi-Fi-N	Wi-Fi-D
Baseline	94 ± 4	76 ± 9	73 ± 11	61 ± 13	44 ± 16
Delay-Only	91 ± 5	87 ± 6	84 ± 7	77 ± 9	64 ± 11
Full Net-Aware	89 ± 4	87 ± 5	85 ± 6	81 ± 7	74 ± 9

Baseline training achieves 94% success in Sim-Clean but drops to 44% in Wi-Fi-D—a 53.2% degradation. Full network-aware training achieves 74% in Wi-Fi-D (17.0% drop from Sim-Clean), a **3.13× reduction in the deployment gap** ($t(9) = 8.4$, $p < 0.001$, Cohen's $d = 1.87$). The smaller absolute effect than CartPole is consistent with MiniGrid's reduced timing sensitivity: delayed actions cause overshooting rather than

catastrophic instability. Delay-only training provides partial robustness (64% in Wi-Fi-D), confirming that stochastic network phenomena require explicit modelling.

8.6.2 Impact of Different Network Pathologies

An ablation study trains CartPole policies under four conditions—latency-only (constant 50 ms), stochastic additional delay ($\Delta t \sim \max(0, \mathcal{N}(0, 40^2))$ ms), loss-only (10% dropout, zero delay), and combined (full network model)—and evaluates all on Real-Wi-Fi-Degraded. Table 8.5 presents the results.

Table 8.5: CartPole Ablation: Mean Episode Return in Real-Wi-Fi-Degraded. Ten seeds per training condition, each evaluated over 50 episodes.

Training Regime	Real-Wi-Fi-Degraded
Baseline (none)	92 ± 54
Latency-Only (50 ms)	275 ± 52
Stochastic Add. Delay (σ=40 ms)	315 ± 47
Loss-Only (10%)	308 ± 49
Combined (full model)	378 ± 41

The ablation reveals a clear hierarchy of network pathology severity. **Stochastic additional delay training** (315 ± 47) outperforms latency-only (275 ± 52), despite both conditions having similar average delay magnitudes. This counterintuitive result has important implications: constant delays allow policies to learn fixed-horizon predictive models ("the state I observe now reflects what happened 50 ms ago; I should plan 50 ms ahead"), whereas stochastic additional delay forces policies to maintain uncertainty estimates over observation freshness. Training under stochastic delay therefore induces more conservative, robust control strategies that hedge against worst-case timing.

Packet loss (308 ± 49) proves similarly detrimental to jitter. When 10% of observations are dropped, policies must infer missing state information or defer actions until fresh observations arrive. Policies trained without loss awareness assume all observations are fresh and trustworthy; when deployed under loss, they act on stale or interpolated observations, leading to control failures. Loss-trained policies learn to detect observation staleness (e.g., via action-observation consistency checks) and adopt conservative strategies when observations are missing.

The **combined training regime** (378 ± 41) significantly outperforms any single-factor training (pairwise t-tests: all $p < 0.01$), demonstrating non-additive interactions between network pathologies. Latency, jitter, and packet loss compound: jittery latency with occasional packet loss creates scenarios where the policy must handle simultaneous timing uncertainty and information gaps. Training under the full joint distribution enables policies to develop integrated coping strategies (e.g., maintaining belief states over

delayed, noisy, and incomplete observations) that single-factor training cannot discover.

8.6.3 Distributed Policy Graph Deployment

Two-level hierarchical architectures for CartPole and MiniGrid illustrate CALF's distributed deployment capabilities. These experiments show that policy graphs trained in simulation can transfer to edge-cloud hardware, and that simple decompositions with time-critical units on edge devices achieve competitive performance whilst exercising the commitment mechanisms of Chapter 5.

CartPole Hierarchical Policy Graph

A two-level CartPole graph decomposes control into an Angle Stabiliser (Unit A, reward $r_A = -|\theta| - 0.1|x|$, deployed on Pi) and a Recentring unit (Unit B, reward $r_B = -|x| - 0.05|\theta| - 0.1|\Delta a|$, deployed on Desktop), with a rule-based manager delegating to Unit A when $|\theta| > 5°$. Table 8.6 shows the distributed deployment achieves 465 ± 24—intermediate between flat-on-Pi (472 ± 21) and flat-on-Desktop (448 ± 28)—whilst achieving 22 ms median latency by keeping time-critical control local. The modest gap relative to flat-on-Pi reflects inter-unit handoff costs, exactly the overhead that the commitment mechanisms of Chapter 5 are designed to amortise.

Table 8.6: CartPole Policy Graph: Performance comparison for distributed deployment. All configurations evaluated over Real-Wi-Fi-Normal network.

Deployment Configuration	Episode Return	E2E Latency (p50/p95)
Flat (Desktop)	448 ± 28	38 ms / 62 ms
Flat (Pi)	472 ± 21	6 ms / 11 ms
Hierarchical (Distributed)	465 ± 24	22 ms / 45 ms

MiniGrid Hierarchical Policy Graph

MiniGrid's natural subgoal structure (find key → unlock door → reach goal) defines two specialist units: Unit K (key policy, deployed on Pi) and Unit G (goal policy, deployed on Desktop), with a rule-based manager switching on `has_key`. The hierarchical deployment achieves 79% success—close to flat-on-Pi (82%) and above flat-on-Desktop (77%)—with the 3-point gap relative to flat-on-Pi not statistically significant ($p = 0.18$). Deploying the time-sensitive key-collection unit locally avoids network round-trips during interactive item manipulation, whilst the goal-navigation unit on Desktop tolerates moderate latency.

These results illustrate the distributed policy graph execution model from Chapter 5 and provide initial evidence that CALF can deploy hierarchical policies across edge-cloud infrastructure.

Table 8.7: MiniGrid Policy Graph: Success rate comparison. All configurations evaluated over Real-Wi-Fi-Normal network.

Deployment Configuration	Success Rate (%)
Flat (Desktop)	77 ± 9
Flat (Pi)	82 ± 7
Hierarchical (Distributed)	79 ± 8

8.6.4 Systems Measurements and Infrastructure Validation

End-to-end latency, throughput, and resource utilisation are measured during distributed policy graph execution to assess CALF's practical feasibility (RQ3). Results indicate that CALF's architecture supports responsive control on resource-constrained edge devices whilst maintaining efficient utilisation of heterogeneous hardware.

End-to-End Latency

Table 8.8 presents latency measurements across network configurations. Latency is measured from environment observation emission to policy action receipt, capturing the full round-trip communication delay.

Table 8.8: End-to-End Latency: Median and 95th percentile latency measured over 1000 environment steps during CartPole policy graph deployment. "Local" indicates co-located environment and policy; "Remote" indicates networked communication.

Configuration	Latency p50 (ms)	Latency p95 (ms)
Local (Pi only)	5.2	9.8
Ethernet (Pi $\leftrightarrow$ Desktop)	8.7	14.3
Wi-Fi-Normal	34.5	68.2
Wi-Fi-Degraded	82.1	152.7

Local execution on Raspberry Pi achieves sub-10 ms latency at p95, validating that edge devices can support responsive control loops. Ethernet deployment adds minimal overhead (8.7 ms median versus 5.2 ms local), reflecting the low latency and near-zero packet loss of wired connections. Wi-Fi-Normal introduces substantial variability (34.5 ms median, 68.2 ms p95), with p95 latency exceeding median by $2\times$ due to jitter and occasional retransmissions. Wi-Fi-Degraded exhibits severe tail latency (152.7 ms p95), demonstrating the worst-case conditions against which network-aware training must be robust.

These measurements validate the network models used in Mode 2 training. Our synthetic Wi-Fi-Normal model ($\mathcal{N}(30, 10^2)$ ms latency, 2% loss) closely matches measured Wi-Fi-Normal (34.5 ms median, implying fitted mean ≈ 34 ms). This alignment confirms

that policies trained in Mode 2 experience representative network conditions, enabling successful transfer to Mode 3 deployment.

Throughput and Resource Utilisation

Table 8.9 reports CPU and memory usage during distributed policy graph execution, demonstrating that CALF's architecture enables balanced workload distribution across heterogeneous hardware.

Table 8.9: Resource Utilisation: Mean CPU and memory usage measured over 10-minute deployment window during CartPole hierarchical policy graph execution. Pi hosts environment and Unit A; Desktop hosts Manager and Unit B.

Device	CPU (%)	Memory (MB)	Throughput (episodes/hour)
Pi	52	310	—
Desktop	18	420	—
System	—	—	1840

The Raspberry Pi operates at 52% average CPU utilisation, indicating headroom for additional workloads or more complex policy networks. Memory usage (310 MB) remains well within the Pi's 4 GB capacity, validating that CALF's binary protocol and efficient serialisation avoid memory bloat. Desktop CPU utilisation is low (18%), reflecting that Manager and Unit B execute lightweight policies; this headroom could be exploited by deploying multiple policy graphs or running compute-intensive strategic planning (e.g., tree search, model-based lookahead) on the cloud server whilst edge devices handle real-time control.

System throughput (1840 episodes/hour) demonstrates that CALF supports high-frequency experimentation. At this rate, evaluating a trained policy over 50 episodes (typical experimental protocol) requires < 2 minutes, enabling rapid iteration during development. For comparison, frameworks that require environment-policy co-location (e.g., Gym running locally) achieve similar throughput but cannot exploit distributed deployment; frameworks that rely on heavyweight RPCs (e.g., gRPC without optimisation) often suffer $5-10\times$ throughput degradation due to serialisation overhead. CALF's custom binary protocol achieves deployment flexibility without sacrificing performance.

8.7 Discussion

8.7.1 Network as an Orthogonal Axis of Sim-to-Real Transfer

Network conditions constitute an **independent dimension of domain randomisation**, orthogonal to physics and visual randomisation. The analogy is direct: just as

physics randomisation samples friction $\sim U(0.3, 0.7)$ to make policies robust to uncertain surfaces, network randomisation samples latency $\sim \mathcal{N}(30 \text{ ms}, 10 \text{ ms}^2)$ to make policies robust to uncertain networks. Both expose the agent to a distribution during training, yielding robustness at deployment. A policy trained with perfect timing may fail catastrophically on a real system with 100 ms lag even if physics are perfectly modelled; the two axes are conceptually and empirically distinct.

The ablation (Section 8.6.2) extends this analogy. Training under constant delay is analogous to sampling friction from a point mass rather than a distribution: the policy adapts to the mean but remains brittle to deviations. Training under stochastic delay forces policies to hedge across a distribution of timing perturbations. The implication is direct: even when the *mean* latency is known, the *variance* must be included in training. CALF provides infrastructure to make network-aware training systematic and reproducible, treating prior delay-aware fixes (e.g., Hwangbo et al. [156] modelling actuator dynamics) as a domain-agnostic methodology rather than robot-specific engineering.

8.7.2 CALF as a Platform for Future Work

Within this thesis, CALF serves as deployment substrate for the distributed-policy work that follows: Chapter 7's efficient edge models address running policy units on resource-constrained hardware; Chapter 5 provides the policy-graph abstraction CALF executes; and Chapter 9's purpose-built USB hardware path relies on the same networking infrastructure. For the research community, natural extensions include trace-based training using recorded real-world network logs, multi-agent settings where inter-agent messages pass through NetworkShim, dynamic computation offloading based on current network state, and integration with delay-correcting algorithms such as DCAC [82].

8.7.3 Limitations

1. Simulated environments. CartPole and MiniGrid are simulated, not physical robots. This allows isolation of network effects but limits ecological validity; future work should validate CALF on physical systems where sensor noise, actuation dynamics, and safety constraints are present.

2. Limited network scenarios. Experiments cover LAN-like conditions only (Ethernet, Wi-Fi within one building). WAN, cellular, and adversarial conditions—each with distinct latency asymmetries and jitter profiles—are not evaluated and may require different training strategies.

3. Simple policy graphs. Distributed deployments use 2-unit decompositions with rule-based managers. Deeper hierarchies (3+ levels), learned option discovery, and end-to-end policy graph training under network constraints remain unexplored; Chapter 5 provides the formalism that such work would require.

4. Offline training and single-agent focus. Policies are trained in simulation then deployed without online adaptation, and CALF currently targets single-agent RL. Online adaptation under deployment-time network conditions and extension to multi-agent coordination under delays are natural next steps.

The core finding—network-aware training reduces the network reality gap by 3–4×—is orthogonal to physics fidelity and plausibly generalises to physical robots, though empirical testing is necessary.

8.7.4 Future Directions

1. Richer environments and modalities. Extending CALF to continuous control (MuJoCo locomotion, manipulation) and vision-based tasks would test network-aware training where bandwidth becomes a first-order constraint alongside latency.

2. Advanced network models. Time-varying conditions (diurnal patterns, congestion), adversarial networks, and trace-based training using cellular or campus Wi-Fi logs would enable policies tuned to specific deployment environments.

3. End-to-end policy graph training. Investigating whether Option-Critic or HIRO-style hierarchies discover temporally extended options naturally robust to communication delays, and learning optimal unit-placement based on communication requirements, remain open problems.

4. Multi-agent and adaptive deployment. Extending CALF to MARL—where inter-agent messages traverse NetworkShim—and developing policies that dynamically offload computation between edge and cloud based on observed network state, represent two directions that would increase practical scope.

8.8 Conclusion

This chapter introduced CALF (Communication-Aware Learning Framework), infrastructure that extends the policy graph framework from Chapter 5 to network-aware distributed execution across heterogeneous hardware. Where Chapter 5 established policy graphs as directed graph structures enabling modular decomposition—with policy units coordinating through hard routing and commitment bounds—this chapter addressed the systems challenge of deploying those policy units across real networks where latency, jitter, and packet loss emerge as first-order constraints.

CALF realises policy graphs as networked services with NetworkShim middleware transparently injecting impairments on graph edges, enabling network-aware training that reduces deployment degradation by 4× (CartPole) and approximately 3× (MiniGrid). Stochastic network phenomena—jitter and packet loss—prove more detrimental than constant latency, challenging the fixed-delay focus of prior delay-aware RL. Illustrative

distributed deployments across Raspberry Pi and desktop hardware demonstrate that hierarchical architectures with time-critical units executing locally maintain competitive performance under network constraints.

These findings establish network conditions as an orthogonal axis of sim-to-real transfer, complementing the physics and visual domain randomisation reviewed in Chapter 4. The architectural patterns of Chapter 3—A320 flight computers distributing responsibility across ELACs and SECs, power grids coordinating IEDs with SCADA—motivate CALF's design: just as engineered systems achieve reliability through hierarchical specialisation, distributed policy graphs partition computation across edge and cloud with accountability through commitment mechanisms. Chapter 7's efficient edge models and Chapter 9's purpose-built hardware path build on this infrastructure, and CALF's progressive deployment modes—pure simulation, simulation with network models, real hardware—stage the path from theory to deployment by treating communication constraints as tractable training objectives rather than deployment obstacles.